REACT JS COMPLETE CHEAT SHEET

300+ CONCEPTS, HOOKS & COMPONENTS EXPLAINED FOR BEGINNERS

- 300+ React Concepts Covered
- All Hooks & Components Explained
- Beginner to Advanced Guide
- Examples with Code

REACT BASICS · COMPONENTS · HOOKS · JSX · ROUTER · CONTEXT API

REACT BASICS

- What is React?
- Create React App
- JSX Syntax
- Rendering Elements
- Props & State
- Event Handling
- Conditional Rendering
- Lists & Keys

```
npx create-react-app my-app
```

2. JSX BASICS — JSX

```
const element = (
  <div className="card">
    <h1>Hello React!</h1>
    <p>JSX makes it easy.</p>
  </div>
);

ReactDOM.createRoot(root)
  .render(element);
```

3. COMPONENTS

Function Component

```
function Welcome=(props) {
  return <h1>Hello, {props.name}!
  </h1>;
}
```

Export

```
export default Welcome;
```

4. PROPS

```
function User(props) {
  return <h2>{props.name}
         ({props.age})</h2>;
}

<User name="John" age={25} />
```

5. STATE

```
function Counter() {
 const [count, setCount] =
 useState(0);
 return (
   <button onClick={() =>
     setCount(count + 1)}>
   Count: {count}
   </button>
};
```

6. USEEFFECT HOOK

```
useEffect(() => {
 document.title = `Count: ${count}`;
 console.log("Mounted");
 return () => {
   console.log("Cleanup");
 };
}, [count]);
```

7. USESTATE HOOK

```
const [name, setName] =
 useState("React");

<input
 value={name}
 onChange={(e) => setName(e.target.value)}
/>
<p>Hello, {name}!</p>
```

8. USECONTEXT

```
const ThemeContext =
 createContext();

const theme = useContext
 (ThemeContext);

return <div>{theme}</div>;
```

9. CONDITIONAL RENDERING

```
{isLoggedIn ? (
  <Dashboard />
) : (
  <Login />
)}
```

10. LISTS & KEYS

```
const items = ['A','B','C'];
<ul>
  {items.map(item => (
    <li key={item}>{item}</li>
  ))}
</ul>
```

11. EVENT HANDLING

```
function handleClick() {
  alert("Button Clicked!");
}
<button onClick={handleClick}>
  Click Me
</button>
```

12. REACT ROUTER

```
<BrowserRouter>
 <Routes>
   <Route path="/" element={<Home />} />
   <Route path="/about" element={<About />} />
 </Routes>
</BrowserRouter>
```

POPULAR HOOKS

useState — Manage state in function components

```
const [state, setState]
= useState(initial);
```

useEffect — Perform side effects in components

```
useEffect(() => {
  // effect
}, [deps]);
```

useContext — Access context values easily

```
const value =
useContext(MyContext);
```

useRef — Access DOM elements or persist values

```
const ref =
useRef(inittalValue);
```

useMemo — Memoize expensive calculations

```
const memo = useMeno(
() => compute(), [depsl);
```

useCallback — Memoize functions to prevent re-renders

```
const fn = useCallback(
() => {}, [depsl);
```

USEFUL TIPS

- Keep components small and reusable
- Use meaningful names for variables & functions
- Use Fragments <> </> to avoid extra DOM
- Optimize performance with React.memo

LEARN REACT JS THE EASY WAY!

SAVE TIME
CODE FASTER
BUILD BETTER

React JS Basics

The building blocks of modern web applications

React

1 React Logo

The React logo represents a library for building user interfaces.

2 JavaScript Library

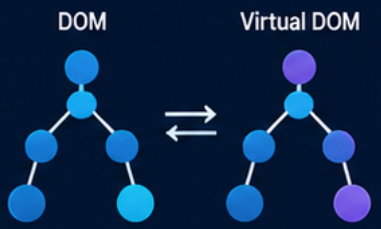
JS

React is a JavaScript library for building interactive and dynamic UIs.

3 Components

Build encapsulated UI pieces called components and combine them.

4 Virtual DOM



React uses Virtual DOM to update the UI efficiently and improve performance.

5 JSX

```
const element = (
  <h1 className="title">
    Hello, React!
  </h1>
);
```

JSX lets you write HTML-like code in JavaScript. It makes UI structure easy.

6 SPA

React is perfect for building Single Page Applications (SPAs).

7 Reusable UI

Components are reusable, maintainable and help in building scalable apps.

8 Fast Rendering

Optimized updates with Virtual DOM and efficient rendering make React super fast.

</> Learn once, write anywhere – **React** makes development faster and easier!

Installation & Setup

Steps and tools to get your React development environment ready.

Node.js

JavaScript runtime built on Chrome's V8 engine.

https://nodejs.org

npm

Node package manager for installing dependencies.

https://www.npmjs.com

npx

Execute packages without installing globally.

`npx <package>`

Vite

Next generation build tool. Fast dev server and blazing builds.

https://vitejs.dev

Create React App

Official CLI for creating React apps with zero configuration.

`npx create-react-app my-app`

Project Structure

```
my-react-app/
├── node_modules/
├── public/
│   └── index.html
├── src/
│   ├── assets/
│   ├── components/
│   ├── App.jsx
│   ├── main.jsx
│   └── index.css
├── .gitignore
├── package.json
├── README.md
└── vite.config.js
```

package.json (Example)

```json
{
  "name": "my-react-app",
  "private": true,
  "version": "0.0.0",
  "type": "module",
  "scripts": {
    "dev": "vite",
    "build": "vite build",
    "preview": "vite preview"
  },
  "dependencies": {
    "react": "^18.3.1",
    "react-dom": "^18.3.1"
  },
  "devDependencies": {
    "vite": "^5.2.0",
    "@vitejs/plugin-react": "^4.2.1"
  }
}
```

Create Project (Examples)

Using Vite (Recommended)

```
# Create React app with Vite
npm create vite@latest my-app -- --template react
cd my-app
npm install
```

Using Create React App

```
npx create-react-app my-app
cd my-app
npm start
```

Using npx (Run without install)

```
npx vite
npx create-react-app my-app
```

Run Development Server

Start the development server and open your app in the browser.

```
# For Vite
npm run dev

# For Create React App
npm start
```

Open in Browser

http://localhost:5173 (Vite)

http://localhost:3000 (CRA)

Tips

- Make sure Node.js is installed (v16+ recommended).
- Use Vite for faster performance and better DX.
- `npm run dev` starts the app in development mode.

JSX Basics

Write HTML-like code in JavaScript.

</> JSX

1. JSX

JSX is a syntax extension for JavaScript.

```jsx
const element = (
  <h1>Hello JSX!</h1>
);
```

2. HTML in JS

JSX looks like HTML, but it is used inside JS.

```jsx
const element = (
  <div>
    <p>This is JSX</p>
  </div>
);
```

3. Expressions

Use curly braces {} to embed JavaScript expressions.

```jsx
const name = "React";
const element = (
  <h2>Hello, {name}!</h2>
);
```

4. Fragments

Use fragments to group multiple elements without extra nodes.

```jsx
const element = (
  <>
    <h1>Title</h1>
    <p>Paragraph</p>
  </>
);
```

5. Comments

Use {/**/} for multi-line comments in JSX.

```jsx
const element = (
  <div>
    {/** This is a
       multi-line comment */}
    <p>Visible Content</p>
  </div>
```

6. Attributes

Use camelCase for most HTML attributes.

```jsx
const element = (
  <input type="text"
    placeholder="Enter name" />
);
```

7. ClassName

Use className instead of class.

```jsx
const element = (
  <div className="container">
    <p className="text">Hello</p>
  </div>
);
```

8. Self Closing Tags

Self-closing tags must be closed with />.

```jsx
const element = (
  <img src="logo.png" alt="Logo" />
  <br />
  <hr />
);
```

Tip: JSX makes your code more readable and easier to write UI in React.

Write Less, Build More!

Components

Building blocks of a React application

1. Functional Component

Components are JavaScript functions that return JSX.

```
function Welcome() {
  return <h1>Hello React!</h1>;
}
```

2. Component Name

Component names must start with a capital letter.

```
function UserCard() {
  return <div>User Card</div>;
}
```

3. Return JSX

Every component must return a single JSX element.

```
function Hello() {
  return (
    <h2>Hello World</h2>
  );
}
```

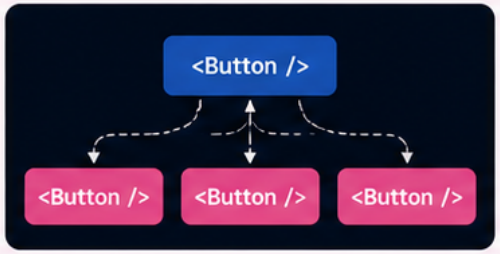
4. Nested Components

Components can be used inside other components.

```
function App() {
  return (
    <div>
      <Header />
      <Content />
    </div>
  );
}
```

5. Reusable

Components are reusable across your application.



6. Export

Export components to use them in other files.

```
function Button() {
  return <button>Click me</button>;
}
export default Button;
```

7. Import

Import components where you want to use them.

```
import Button from './Button';
function App() {
  return <Button />;
}
```

8. Props

Props are inputs passed to components for dynamic data.

Parent Component

```
<User name="John" age={25} />
```

Child Component

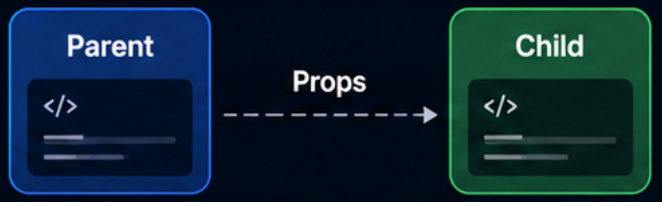
```
function User(props) {
  return <p>{props.name} - {props.age}</p>;
}
```

Components make your code modular, maintainable and easy to scale.

Think: **Build small, use everywhere!**

Props

Props are used to pass data from one component to another.



1. Props

Props (short for properties) are read-only inputs passed to components.

```
function Greeting(props) {
  return <h1>Hello, {props.name}!</h1>;
}

<Greeting name="React" />
```

2. Passing Data

Send data from parent to child using props.

```
function Parent() {
  return <Child message="Hello!" />;
}
function Child(props) {
  return <p>{props.message}</p>;
}
```

3. Children

Props can be used to pass children (nested elements).

```
function Wrapper({ children }) {
  return <div className="box">
    {children}
  </div>;
}
<Wrapper><p>I'm a child</p></Wrapper>
```

4. Default Props

Provide default values for props.

```
function Button({ text = "Click Me" }) {
  return <button>{text}</button>;
}
<Button />
<Button text="Submit" />
```

5. Object Props

Pass objects as props to components.

```
const user = {
  name: "John",
  age: 25,
  city: "Delhi"
};

<UserProfile user={user} />
```

6. Array Props

Pass arrays as props and render list items.

```
const items = ["React", "Vue", "Angular"];
function List({ items }) {
  return (
    <ul>{items.map(item =>
      <li key={item}>{item}</li>)}</ul>
  );
}
<List items={items} />
```

7. Function Props

Pass functions as props to handle events.

```
function Parent() {
  const handleClick = () => alert("Hi!");
  return <Child onClick={handleClick} />;
}
function Child({ onClick }) {
  return <button onClick={onClick}>Click</button>;
}
```

8. Destructuring

Use destructuring to extract props easily.

```
function Profile({ name, age, city }) {
  return (
    <p>{name} is {age} from {city}</p>
  );
}
```

Tip: Props flow down from parent to child. They are immutable (read-only) and help build dynamic and reusable components.

Remember: Never modify props inside the child component.

State

State lets you add dynamic behavior to your components.

1. State

State is a built-in React object that holds data which can change over time.

```jsx
import { useState } from 'react';

function Counter() {
  const [count, setCount] =
    useState(0);

  return <h1>Count: {count}</h1>;
}
```

2. useState

useState is a Hook that returns a stateful value and a function to update it.

```jsx
const [state, setState] = useState(
  initialValue
);

// state    -> current value
// setState -> function to update
```

3. Update State

Use the updater function to change the state. Do not modify state directly.

```jsx
const [count, setCount] = useState(0);

const increment = () => {
  setCount(count + 1); // ✅ Correct
  // count = count + 1; // ❌ Wrong
};
```

4. Re-render

When state updates, React re-renders the component and reflects the changes in the UI.

React batches state updates for better performance.

5. Primitive State

Store simple values like string, number, boolean in state.

```jsx
const [name, setName] = useState("John");
const [age, setAge] = useState(25);
const [isActive, setIsActive] =
  useState(true);

return (
  <p>{name} - {age} - {isActive ? 'Yes' : 'No'}</p>
);
```

6. Object State

Store multiple values in an object. Update using spread operator.

```jsx
const [user, setUser] = useState({
  name: "John",
  age: 25,
  city: "Delhi"
});

const updateCity = () => {
  setUser({ ...user, city: "Mumbai" });
};
```

7. Array State

Store lists in state and update them immutably.

```jsx
const [items, setItems] = useState(["A", "B"]);

const addItem = () => {
  setItems([...items, "C"]);
};

const removeItem = (item) => {
  setItems(items.filter(i => i !== item));
};
```

8. Immutable Update

Always treat state as read-only and update it immutably.

❌ Mutable (Wrong)

```jsx
user.name = "Mike";
items.push("D");        // Don't do this!
```

✅ Immutable (Correct)

```jsx
setUser({ ...user, name: "Mike" });
setItems([...items, "D"]);
                        // Do this instead!
```

Tip: State changes trigger re-renders. Keep state minimal and update it immutably for predictable and efficient apps.

Best Practice: Keep related data together, update immutably, and avoid unnecessary state.

Event Handling

Handle user interactions in React applications.

User Interaction → Event Handler → Your Logic → UI Updates

1. onClick

Triggered when an element is clicked.

```jsx
function ButtonClick() {
  const handleClick = () => {
    alert('Button Clicked!');
  };

  return <button onClick={handleClick}>
    Click Me
  </button>;
}
```

2. onChange

Triggered when the value of an input changes.

```jsx
function InputChange() {
  const [text, setText] = useState('');

 return (
   <input
     type="text"
     value={text}
     onChange={(e) => setText(e.target.value)}
     placeholder="Type something..."
   />
 );
}
```

3. onSubmit

Triggered when a form is submitted.

```jsx
function FormSubmit() {
  const handleSubmit = (e) => {
    e.preventDefault();
    alert('Form Submitted!');
  };

  return (
    <form onSubmit={handleSubmit}>
     <button type="submit">Submit</button>
    </form>
  );
}
```

4. onKeyDown

Triggered when a key is pressed down.

```jsx
function KeyDown() {
  const handleKeyDown = (e) => {
    console.log('Key:', e.key);
  };

  return (
    <input
      type="text"
      onKeyDown={handleKeyDown}
      placeholder="Press any key..."
    />
);});
```

5. onMouseEnter

Triggered when the mouse enters an element.

```jsx
function MouseEnter() {
  const handleEnter = () => {
    console.log('Mouse entered!');
  };

 return (
   <div
    onMouseEnter={handleEnter}
    style={{ padding: '20px', background: '#ffeaa7' }}
   >
    Hover over me
   </div>
 );
}
```

6. Arrow Function

Use arrow functions for concise event handlers.

```jsx
function ArrowExample() {
  return (
    <button
     onClick={() => alert('Hello!')}
     style={{ padding: '10px 20px' }}
    >
     Say Hello
    </button>
  );
}
```

Arrow functions are concise and bind 'this' automatically.

7. Event Object

Access event object to get details about the event.

```jsx
function ShowEvent(e) {
  console.log('Event Type:', e.type);
  console.log('Target:', e.target);
}

return <button onClick={ShowEvent}>
  Log Event Info
 </button>;
```

The event object contains useful information like type, target, key, etc.

8. Prevent Default

Use e.preventDefault() to prevent the default browser behavior.

```jsx
function HandleSubmit(e) {
  e.preventDefault(); // Prevent page reload
  alert('Form submitted without reload!');
}

return (
  <form onSubmit={HandleSubmit}>
   <input type="text" placeholder="Name" />
   <button type="submit">Submit</button>
  </form>
);
```

Always use preventDefault() in form submissions handled by React.

Pro Tip: React uses camelCase for events (onClick, onChange, etc.)
All event handlers receive an event object as the first argument.

Remember: Events in React are named using camelCase and passed as functions, not strings.

Conditional Rendering

Render different UI based on different conditions.

Evaluate Condition → Choose UI → Render Result

1 if

Use JavaScript if statement for complex conditions.

```
function Greeting({ isLoggedIn }) {
  if (isLoggedIn) {
    return <h1>Welcome Back!</h1>;
  } else {
    return <h1>Please Login</h1>;
  }
}
```

Logged In: Welcome Back!

Logged Out: Please Login

2 Ternary

Use ternary operator for simple conditions in JSX.

```
function User({ isAdmin }) {
  return (
    <div>
      {isAdmin ? <span>Admin</span> :
        <span>User</span>}
    </div>
  );
}
```

isAdmin = true: Admin

isAdmin = false: User

3 && Operator

Use && to render UI only when condition is true.

```
function Notification({ show }) {
  return (
    <div>
      {show && <p className="alert">
        New message received!</p>}
    </div>
  );
}
```

show = true: New message received!

show = false: (Nothing rendered)

4 Null

Return null to render nothing.

```
function Nothing({ visible }) {
  if (!visible) return null;
  return <div>Now you see me!</div>;
}
```

visible = true: Now you see me!

visible = false: (Nothing rendered)

5 Optional UI

Render parts of UI that are optional.

```
function Profile({ user }) {
  return (
    <div>
      <h2>{user.name}</h2>
      {user.bio && <p>{user.bio}</p>}
      {user.website && (
        <a href={user.website}>Website</a>
      )}
    </div>
  );
}
```

John Doe
Frontend Developer
johndoe.dev

John Doe
(No bio or website)

6 Loading

Show loading UI while data is being fetched.

```
if (loading) {
  return <div className="loading">
    <div className="spinner"></div>
    <p>Loading...</p>
  </div>;
}

return <Data />;
```

Loading...
Loading data, please wait...

7 Error State

Show error message when something goes wrong.

```
if (error) {
  return (
    <div className="error">
      <h3>Something went wrong!</h3>
      <p>{error.message}</p>
    </div>
  );
}
```

Something went wrong!
Failed to fetch data. Please try again.

8 Empty State

Show empty state when there is no data.

```
if (data.length === 0) {
  return (
    <div className="empty">
      <img src="/empty.svg" alt="empty" />
      <p>No items found.</p>
    </div>
  );
}
```

No items found.
Looks like there is nothing here.

Pro Tip Keep your conditions simple in JSX. Extract complex logic outside the return for better readability.

Best Practice Handle all states: Loading, Empty and Success for better UX.

Remember UI should respond to the state of your data and user actions.

Lists & Keys

Render collections of data efficiently in React.

Data Array → map() → Unique Key → Rendered List

1 map()

Use map() to loop through array and return JSX.

```
const items = ['Apple', 'Banana', 'Mango'];

const list = items.map((item) => (
  <li key={item}>{item}</li>
));

return <ul>{list}</ul>;
```

Output

- Apple
- Banana
- Mango

2 Keys

Keys help React identify which items changed, added or removed.

```
const list = items.map((item) => (
  <li key={item.id}>{item.name}</li>
));
```

Good

Use stable and unique keys.

`key={item.id}`

Bad

Avoid using index as key.

`key={index}`

3 Unique ID

Always use a unique id from your data as key.

```
const users = [
  { id: 1, name: 'John' },
  { id: 2, name: 'Sara' },
  { id: 3, name: 'Mike' }
];

const list = users.map(user => (
  <li key={user.id}>{user.name}</li>
));
```

Data

- id: 1, name: 'John'
- id: 2, name: 'Sara'
- id: 3, name: 'Mike'

4 Rendering List

Render list of elements in the UI.

```
return (
  <ul className="user-list">
    {users.map(user => (
      <li key={user.id} className="user-item">
        {user.name}
      </li>
    ))}
  </ul>
);
```

UI Preview

- John
- Sara
- Mike

5 Filter

Use filter() to show items that match a condition.

```
const activeUsers = users.filter(user =>
  user.isActive
);

const list = activeUsers.map(user => (
  <li key={user.id}>{user.name}</li>
));
```

Show only active users.

`filter(user => user.isActive)`

6 Sort

Use sort() to sort items before rendering.

```
const sortedUsers = [...users].sort((a, b) =>
  a.name.localeCompare(b.name)
);

const list = sortedUsers.map(user => (
  <li key={user.id}>{user.name}</li>
));
```

Sort by name (A → Z).

`sort((a, b) => a.name.localeCompare(b.name))`

7 Nested List

Render lists inside another list.

```
const categories = [
  { id: 1, title: 'Fruits', items: ['Apple','Banana'] },
  { id: 2, title: 'Veggies', items: ['Carrot','Potato'] }
];

return (
  <ul>
    {categories.map(cat => (
      <li key={cat.id}>
        <strong>{cat.title}</strong>
        <ul>
          {cat.items.map(item => (
            <li key={item}>{item}</li>
          ))}
        </ul>
      </li>
    ))}
  </ul>
);
```

- Fruits
  - Apple
  - Banana
- Veggies
  - Carrot
  - Potato

8 Dynamic List

Add, remove or update items dynamically.

```
const [items, setItems] = useState(['Task 1']);

const addItem = () => {
  setItems([...items, 'Task ${items.length + 1}']);
};

const removeItem = (id) => {
  setItems(items.filter((_, index) => index !== id));
};

return (
  <div>
    <ul>
      {items.map((item, index) => (
        <li key={index}>
          {item} <button onClick={() => removeItem(index)}>X</button>
        </li>
      ))}
    </ul>
    <button onClick={addItem}>Add Item</button>
</div>
);
```

Tasks

- Task 1 ×
- Task 2 ×
- Task 3 ×

Add Item

Pro Tip: Always use unique and stable keys. Avoid index as key for better performance.

Best Practice: Use meaningful IDs from your data. It helps React update UI efficiently.

Remember: Keys should be unique among siblings, not globally.

Forms

Handle user input and form submission in React.

User Input → Handle State → Validate Data → Submit Form

1 Controlled Input

Value is controlled by React state.

```jsx
const [name, setName] = useState('');
return (
  <input
    type="text"
    value={name}
    onChange={(e) => setName(e.target.value)}
    placeholder="Your name"
);>
```

Name

John Doe

- React controls the value
- Every change updates state

2 Uncontrolled

Value is handled by the DOM using ref.

```jsx
const inputRef = useRef();
const handleSubmit = (e) => {
  e.preventDefault();
  alert(inputRef.current.value);
};

return <input type="text" ref={inputRef} />;
```

Name

John Doe

- DOM controls the value
- Use ref to access value

3 Input

Text input field.

```jsx
return (
  <input
    type="text"
    value={email}
    onChange={(e) => setEmail(e.target.value)}
    placeholder="Enter email"
  />
);
```

Email

Enter email

- type="text"
- type="email"
- type="number"
- type="password"

4 Textarea

Multi-line text input.

```jsx
return (
  <textarea
    value={message}
    onChange={(e) => setMessage(e.target.value)}
    rows={4}
    placeholder="Write your message..."
  />
);
```

Message

Write your message...

- rows to set height
- Perfect for long text

5 Select

Dropdown select input.

```jsx
return (
  <select value={city} onChange={(e) => setCity(e.target.value)}>
    <option value="">Select City</option>
    <option value="delhi">Delhi</option>
    <option value="mumbai">Mumbai</option>
    <option value="bangalore">Bangalore</option>
  </select>
);
```

City

Delhi

- value controls selected option
- onChange updates state

6 Checkbox

Select one or more options.

```jsx
const [agree, setAgree] = useState(false);
return (
  <label>
    <input type="checkbox" checked={agree}
      onChange={(e) => setAgree(e.target.checked)} />
    I agree to the terms & conditions
  </label>
);
```

I agree to the terms & conditions

- checked controls state
- onChange updates state

7 Radio

Select only one option.

```jsx
const [gender, setGender] = useState('');
return (
  <div>
    <label>
      <input type="radio" value="male" checked={gender=='male'}
        onChange={(e) => setGender(e.target.value)} /> Male
    </label>
    <label>
      <input type="radio" value="female" checked={gender=='female'}
        onChange={(e) => setGender(e.target.value)} /> Female
    </label>
  </div>
);
```

Male Female

- Only one can be selected
- Use same name for group

8 Validation

Validate user input before submit.

```jsx
const [email, setEmail] = useState('');
const [error, setError] = useState('');
const handleSubmit = (e) => {
  e.preventDefault();
  if(!email.includes('@')) {
    setError('Please enter a valid email');
    return;
  }
  // submit form
};
```

Email

john@example.com

- Valid email
- Please enter a valid email

Pro Tip Use controlled components for most cases. Always validate input to improve user experience.

Best Practice Keep form state minimal and validate on change or on submit.

Remember Prevent default behavior on submit. Show clear error messages to users.

useState Hook

The most basic Hook to add state to your components.

useState → Declare State → Setter → Update State → Component → Re-renders → UI → Updated

```
const [state, setState] = useState(initialValue);
```

- **state** — The current value
- **setState** — Function to update the value

How it works

Initial Value (Set once) → State Changes (Using Setter) → Component Re-renders (UI Updates)

1 useState

Basic syntax of useState Hook.

```jsx
import { useState } from 'react';
function Example() {
  const [name, setName] = useState('John');
  return (
    <div>
      <p>Name: {name}</p>
      <button onClick={() => setName('Mike')}>
        Change Name
      </button>
    </div>
  );
}
```

Output: Name: Mike

2 Setter

Setter updates the state value.

```jsx
const [count, setCount] = useState(0);

const increment = () => {
  setCount(count + 1);
};
const decrement = () => {
  setCount(count - 1);
};
return (
  <div>
    <p>Count: {count}</p>
    <button onClick={increment}>+1</button>
    <button onClick={decrement}>-1</button>
  </div>
);
```

Calling the setter triggers a re-render.
initial value.

4 Counter

Classic counter example.

```jsx
const [count, setCount] = useState(0);
return (
  <div className="counter">
    <h2>{count}</h2>
    <button onClick={() => setCount(count + 1)}>
      Increment
    </button>
    <button onClick={() => setCount(count - 1)}>
      Decrement
    </button>
  </div>
);
```

UI Preview

− 5 +

5 Boolean

Manage boolean (true / false) state.

```jsx
const [isVisible, setIsVisible] = useState(false);
return (
  <div>
    <p>{isVisible ? 'Visible' : 'Hidden'}</p>
    <button onClick={() => setIsVisible(!isVisible)}>
      Toggle
    </button>
  </div>
);
```

UI Preview

Status: Hidden [Toggle]

6 Array

Manage array state.

```jsx
const [items, setItems] = useState(['Apple', 'Banana']);

const addItem = (item) => {
  setItems([...items, item]);
};

return (
  <div>
    <ul>
      {items.map((item, index) => (
        <li key={index}>{item}</li>
      ))}
    </ul>
    <button onClick={() => addItem('Mango')}>Add Mango</button>
  </div>
);
```

UI Preview

• Apple • Banana • Mango [Add Mango]

7 Object

Manage object state.

```jsx
const [user, setUser] = useState({
  name: 'John',
  age: 25
});

const updateAge = () => {
  setUser({ ...user, age: user.age + 1 });
};
return (
  <div>
    <p>{user.name} - {user.age} years old</p>
    <button onClick={updateAge}>Increase Age</button>
  </div>
);
```

UI Preview

John - 25 years old [Increase Age]

8 Update Function

Use function update for previous state.

```jsx
const [count, setCount] = useState(0);
const increment = () => {
  setCount(prev => prev + 1);
};
// Useful for multiple updates
const incrementTwice = () => {
  setCount(prev => prev + 1);
  setCount(prev => prev + 1);
};
return <button onClick={increment}>+1</button>;
```

Function update ensures you always get the latest state value.

Pro Tip — Keep state as simple as possible. Split complex state into multiple useState.

Best Practice — Avoid directly mutating state. Always create new copies (spread operator).

Remember — State updates are asynchronous. Don't rely on the state value right after setting it.

useEffect Hook

Perform side effects in function components.

useEffect Run Effect → Dependencies Decide When → Cleanup Clean Up → Component Life Cycle

useEffect(() => {}, [dependencies])

Runs after the component renders.

What it does?

useEffect lets you perform side effects like fetching data, subscribing, or manually changing the DOM.

When it runs?

After render (mount, update or unmount) based on the dependencies.

1 useEffect

Basic syntax of useEffect Hook.

```
import { useEffect } from 'react';

function Example() {
  useEffect(() => {
    console.log('✅ Effect runs after render');
  });

  return <div>Hello useEffect</div>;
}
```

Runs after every render by default.

2 Dependencies

Control when the effect runs.

```
// Runs only on mount
useEffect(() => {
  console.log('Runs once');
}, []);

// Runs when 'count' changes
useEffect(() => {
  console.log('Count changed');
}, [count]);
```

[] = Run once (on mount only)
[count] = Run when count changes

3 Cleanup

Clean up before unmount or before next effect.

```
useEffect(() => {
  const timer = setInterval(() => {
    console.log('⏱ Timer running');
  }, 1000);

  return () => {
    clearInterval(timer);
    console.log('🧹 Cleanup: Timer cleared');
  };
}, []);
```

Cleanup runs before next effect or when component unmounts.

4 API Call

Common use case: fetch data from API.

```
useEffect(() => {
  const fetchData = async () => {
    try {
      const res = await fetch('https://api.example.com/data');
      const data = await res.json();
      setData(data);
    } catch (error) {
      console.error('API Error:', error);
    }
  },

  fetchData();
}, []);
```

Use [] to call API only once on mount.

5 Lifecycle Flow

Understand when useEffect runs.

Mount (First Render) → Update (State/Props Change) → Unmount (Component Removed)

Stage	What Happens
Mount	Component renders → useEfect runs
Update	Dependencies change → cleanup (old) → useEffect runs
Unmount	Cleanup runs → component is removed

6 Mount

Runs when component mounts.

```
useEffect(() => {
  console.log('🚀 Component Mounted');

  // Logic here
}, []);
```

Timeline

Mount (Render) — useEffect Runs (After Render) — Update (Optional)

[] ensures this runs only once when component is added to the DOM.

7 Unmount

Runs when component is removed.

```
useEffect(() => {
  console.log('Component Mounted');

  return () => {
    console.log('🧹 Component Unmounted');
    // Cleanup logic here
  };
}, []);
```

Timeline

Mount — Update — Unmount (Cleanup Runs Before Removal)

Cleanup function runs once before the component is unmounted.

8 Common Patterns

Useful patterns with useEffect.

- **Fetch Data** — Call API and set data.
- **Event Listener** — Add event listener and remove on cleanup.
- **Timer / Interval** — Set intervals or timeouts.
- **Subscribe / Unsubscribe** — Subscribe to a service and unsubscribe.

Pro Tip Always specify dependencies to avoid unwanted re-renders or bugs.

Best Practice Keep effects small, focused and clean up what you set up.

Remember useEffect doesn't run during server-side rendering. It only runs in the browser.

useRef Hook

Access DOM elements or persist mutable values across renders.

useRef() Create Ref → .current Access Value → Persist Across Renders → UI Updated

What is useRef?

useRef returns a mutable ref object whose .current property persists for the full lifetime of the component.

```
import { useRef } from 'react';

const myRef = useRef(initialValue);
// myRef.current
```

Structure

myRef → (current: initialValue)

1 DOM Access

Access DOM nodes directly.

```
const inputRef = useRef(null);
return (
  <input
    ref={inputRef}
    type="text"
    placeholder="Your name"
  />
);
```

Your name — inputRef.current → <input />

Use ref to interact with DOM nodes directly.

2 Focus Input

Focus an input field programmatically.

```
const inputRef = useRef(null);

const handleFocus = () => {
  inputRef.current.focus();
};

return (
  <>
    <input ref={inputRef} />
    <button onClick={handleFocus}>Focus</button>
  </>
);
```

Click button to focus input — Focus

Useful for focus management.

3 Mutable Value

Store mutable values without causing re-renders.

```
const countRef = useRef(0);

const handleClick = () => {
  conutRef.current += 1;
  console.log(countRef.current);
};

// Updating ref does NOT re-render component
```

countRef.current — 0 → 1 → 2 → 3 ...

Value changes without re-rendering UI.

4 Current

Access & update the .current property.

```
const valueRef = useRef('Hello');

console.log(valueRef.current); // Hello

valueRef.current = 'World';

console.log(valueRef.current); // World
```

.current — Get or set the current value stored in the ref.

5 Reference

The ref object reference stays the same across renders.

```
const refObj = useRef(0);

useEffect(() => {
  console.log(refObj === prevRef.current);
  prevRef.current = refObj;
});

const prevRef = useRef(null); // holds previous ref
// Always troe
```

The ref object is created once and persists across renders.

6 Persist Data

Persist any data across renders without re-rendering.

```
const timerRef = useRef(null);

const startTimer = () => {
  timerRef.current = setTimeout(() => {
    console.log('Time up!');
  }, 2000);
};
```

Store timers, intervals, previous values, IDs, or any data you want to persist.

7 Scroll

Access and control scroll position.

```
const boxRef = useRef(null);

const scrollToBottom = () => {
  boxRef.current.scrollTop =
    boxRef.current.scrollHeight;
};
return <div ref={boxRef} className="scroll-box" />;
```

Item 1
Item 2
Item 3
...
Item 20

Scroll to Bottom

boxRef.current →

8 When to Use useRef?

- Direct DOM access — Focus, scroll, measurements, etc.
- Store mutable values — That don't affect UI.
- Persist data between renders — Timers, intervals, previous values.
- Avoid unnecessary re-renders — Changing ref doesn't trigger re-render.

Pro Tip Don't use useRef to store values that affect what's shown on the screen.

Best Practice Use useRef for and DOM interactions.

Remember useRef returns the same object on every render. Only .current value can change.

useMemo Hook

Memoize expensive values and optimize performance.

useMemo() Memoize Value → Cache Value (Until deps change) → Skip Re-calculation (Reduce Work) → Better Performance (Faster UI)

What is useMemo?

useMemo is a React Hook that returns a memoized value. It only recalculates the value when its dependencies change.

```
const memoizedValue = useMemo(
  () => calculateSomething(a, b),
  [a, b] // dependencies
);
```

Syntax

useMemo(**Factory Function** , **Dependencies Array**)

- Factory Function: Return the computed value
- Dependencies Array: Re-calculate only when deps change

1 useMemo

Basic usage of useMemo.

```
import { useMemo } from 'react';

function Example({ a, b }) {
  const sum = useMemo(() => {
    console.log('Calculating sum...');
    return a + b; // Expensive calculation
  }, [a, b]); // Re-calculate when a or b changes

  return <div>Sum: {sum}</div>;
}
```

How it works

The function inside useMemo runs only when dependencies change. Otherwise, cached value is returned.

2 Optimization

Avoids unnecessary recalculations.

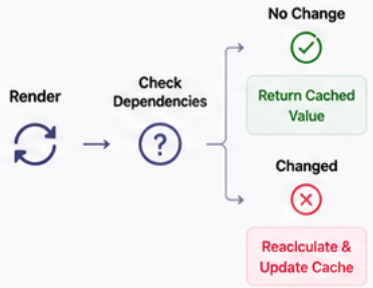
```
function ExpensiveComponent({ list }) {

  const filtered = useMemo(() => {
    console.log('Filtering list...');
    // Simulate expensive operation
    return list.filter(item => item.active);
  }, [list]); // Runs only when list changes

  return <List items={filtered} />;
}
```

useMemo prevents expensive operations on every render.

3 Memoization

Caches the computed value.

How Memoization Works



Value is cached until dependencies change.

4 Expensive Calculation

Perfect for heavy computations.

```
const factorial = useMemo(() => {
  console.log('Calculating factorial...');
  // Expensive calculation
  let result = 1;
  for (let i = 2; i <= n; i++) {
    result *= i;
  }
  return result;
}, [n]);
```

Use useMemo for operations like:

- Loops & calculations
- Filtering large lists
- Complex data transformations

5 Dependencies

Control when the memoized value updates.

```
const value = useMemo(() => compute(a, b), [a, b]);
```

Correct

- Include all values used inside the function
- Keeps data in sync

Incorrect

- Missing dependencies can lead to stale values
- Causes bugs

ESLint (react-hooks/exhaustive-deps) helps you add the right dependencies.

6 Cache

Stores the last computed value.

Render	Dependencies	Action	Result
1st Render	[a, b] = [1, 2]	Calculate	Store in cache
2nd Render	[a, b] = [1, 2]	Use Cache	Return cached value
3rd Render	[a, b] = [2, 2]	Recalculate	Update cache

useMemo caches the result based on dependencies.

7 Performance

Improves app performance.

Without useMemo

- Recalculates on every render
- More CPU usage
- Slower UI

More Renders = More Work

With useMemo

- Recalculates only when needed
- Less CPU usage
- Faster UI

Fewer Renders = Better Performance

Better performance leads to better user experience.

8 Re-render

Prevents unnecessary re-renders.

```
const user = useMemo(() => ({
  name: name,
  age: age,
}), [name, age]);

// Pass to child component
<ChlidComponent user={user} />
```

Stable reference prevents child components from re-rendering.

Pro Tip Don't overuse useMemo. Use it only for expensive calculations.

Best Practice Keep dependencies correct. Measure performance before optimizing.

Remember Premature optimization can make code complex.

useCallback Hook

Memoize functions and prevent unnecessary re-creations.

useCallback() Memoize Function → () => {} Stable Reference → Pass to Child Component → Better Performance Fewer Re-renders

What is useCallback?

useCallback returns a memoized function. The function is recreated only when its dependencies change.

```
const memoizedFn = useCallback(
  () => {
    doSomething(a, b);
  },
  [a, b] // dependencies
);
```

Syntax

useCallback(Callback Function , Dependencies Array)

Function you want to memoize

Re-created only when deps change

1 Basic Usage

Memoize a callback function.

```
const handleClick = useCallback(() => {
  console.log('Button clicked!');
}, []);

// Pass to child component
return <Child onAction={handleClick} />;
```

With [], the function is created once on mount and stays the same.

2 Dependencies

Function updates when deps change.

```
const handleSubmit = useCallback(() => {
  submitForm(data);
}, [data]);

// New function only when `data` changes
// Keeps reference stable otherwise
```

- Correct: [data]
- Incorrect: [] (stale data)

3 Child Component

Prevent unnecessary re-renders.

Parent Component — Pass Callback as Prop → Child Component

Without useCallback

New function on every render → Child re-renders

With useCallback

Same function reference → Child skips re-render

4 Optimization

useCallback improves performance.

- Avoids recreating functions
- Prevents unnecessary child re-renders
- Useful in large or complex apps
- Helps with React.memo and PureComponent

Use it when:
You pass functions to optimized child components.

5 Stable Function Reference

Returns the same function reference.

```
const handle = useCallback(() => {
  console.log('Hello');
}, []);

console.log(handle === handle); // true
// Same reference across renders
```

Stable reference helps React skip unnecessary work.

6 Performance Example

useCallback vs normal function.

Without useCallback

```
const handle = () => {
  doSomething();
};

// New function on every render
// Child re-renders every time
```

More Re-renders
Lower Performance

With useCallback

```
const handle = useCallback(() => {
  doSomething();
}, []);

// Same function reference
// Child re-renders only when needed
```

Fewer Re-renders
Better Performance

7 Real World Example

Optimizing a button click handler.

```
const Counter = () => {
  const [count, setCount] = useState(0);

  const handleIncrement = useCallback(() => {
    setCount(c => c + 1);
  }, []);

  return <ChildButton onClick={handleIncrement} />;
};
```

ChildButton won't re-render unless handleIncrement changes.

8 Summary

Key takeaways.

- useCallback memoizes functions. Re-created only when dependencies change.
- Helps prevent unnecessary re-renders in child components.
- Improves performance in large or complex applications.
- Use with React.memo for maximum benefit.

Pro Tip Don't overuse useCallback. Use it only when it solves a real performance issue.

Best Practice Always include all dependencies. Missing deps can cause bugs (stale closures).

Remember Stable functions + React.memo = Better Performance!

useContext Hook

Access global data easily and avoid prop drilling.

Create Context
createContext()

Provide Value
<Provider value={}>

Use in Component
useContext()

Global Access
Without Prop Drilling

What is useContext?

useContext is a React Hook that lets you access context data without passing props through every level of the component tree.

```jsx
import { useContext } from 'react';
import { MyContext } from './MyContext';

function MyComponent() {
  const value = useContext(MyContext);
  return <div>{value}</div>;
}
```

useContext reads the closest matching Provider value above it in the tree.

1 Context

Create a context with createContext().

```jsx
import { createContext } from 'react';

// Create Context
export const AppContext = createContext();

// Default value (optional)
export const UserContext = createContext({
  user: null,
  setUser: () => {}
});
```

Context holds data that can be shared across the component tree.

2 Provider

Wrap components with Provider and pass the value.

```jsx
import { AppContext } from './AppContext';

function App() {
 const data = {
  theme: 'dark',
  language: 'en',
  user: { name: 'John', role: 'Admin' }
 };

return (
  <AppContext.Provider value={data}>
   <MainApp />
  </AppContext.Provider>
 );
}
```

Provider makes the value available to all child components.

3 Consumer (useContext)

Access context value in any child component.

```jsx
import { useContext } from 'react';
import { AppContext } from './AppContext';

function Header() {
 const { theme, user } = useContext(AppContext);

return (
   <header className={theme}>
     <h1>Welcome, {user.name}</h1>
   </header>
 );
}
```

Any component can consume the context without prop drilling.

4 Global State

Perfect for global data like theme, user, settings, etc.

- Theme
- Language
- User Data
- App Settings

Share and update global data from one place.

5 Theme Example

Change theme across the app.

```jsx
const ThemeContext = createContext();

// In App.js
<ThemeContect.Provider value={theme}>
  <App />
</ThemeContect.Provider>

// In any component
const { theme, setTheme } = useContext(ThemeContext);
```

Light — Dark

6 Language Example

Support multi-language app.

```jsx
const LanguageContext = createContext();

// Provider
<LanguageContext.Provider value={lang}>
  <App />
</LanguageContext.Provider>

// Consumer
const { lang, setLang } = useContext(LanguageContext);
```

English EN | हिंदी HI | ਪੰਜਾਬੀ PA

7 User Data Example

Access logged-in user anywhere.

```jsx
const UserContext = createContext();

// Provider
<UserContect.Provider value={user}>
  <App />
</UserContext.Provider>

// Consumer
const { user, logout } = useContext(UserContext);
```

John Doe
john@example.com
Logout

8 Avoid Prop Drilling

No need to pass props through every level.

Without useContext (Prop Drilling)

App → Layout → Page → Header → Button
(props passed at each step)

With useContext

App (Provider) → Button (Consumer)

9 Best Practices

- Use context for global data, not for all state.
- Split contexts for unrelated data to improve performance.
- Memoize provider value to prevent unnecessary re-renders.
- Keep context value shape stable.

Pro Tip useContext re-renders when the context value changes, so optimize the value.

Best Practice Use it for low-frequency updates like theme, language, user, settings.

Remember useContext makes your code cleaner, scalable and easier to maintain.

useReducer Hook

Manage complex state logic with reducer functions.

useReducer() Manage State → Reducer State Logic → Dispatch Send Action → Updated State New State

What is useReducer?

useReducer is a React Hook that manages complex state logic with a reducer function. It is an alternative to useState for state that has multiple sub-values or complex update logic.

```
import { useReducer } from 'react';
function MyComponent() {
  const [state, dispatch] = useReducer(
    reducer,     // reducer function
    initialState  // initial state
  );
}
```

Best for complex state, predictable logic and state transitions.

1 useReducer

Basic syntax of useReducer.

```
const [state, dispatch] = useReducer(
  reducer,
  initialState,
  init   // (optional) lazy initializer
));
```

state	Current state returned by the reducer.
dispatch	Function to send an action.
reducer	Function that returns the next state.
initialState	Default state value.

2 Reducer

A reducer is a pure function.

```
function reducer(state, action) {
  switch (action.type) {
    case 'INCREMENT':
      return { ...state, count: state.count + 1 };
    case 'DECREMENT':
      return { ...state, count: state.count - 1 };
    default:
      return state;
  }
}
```

Takes (state, action) and returns the next state. No side effects!

3 Dispatch

Send actions to update state.

```
dispatch({ type: 'INCREMENT' });

dispatch({ type: 'DECREMEENT' });

dispatch({
  type: 'SET_COUNT',
  payload: 10
});
```

Dispatch triggers the reducer with the action and updates the state.

4 Action

Actions describe what happened.

```
// Simple action
{ type: 'INCREMENT' }

// With payload
{
  type: 'SET_COUNT',
  payload: 10
}
```

An action is a plain object with a type and optional payload.

5 Initial State

Define the initial state.

```
const initialState = {
  count: 0,
  user: null,
  loading: false,
  error: null
};
```

Initial state can be a value or created by an initializer function.

6 Complex State

Perfect for complex state objects.

```
const initialState = {
  users: [],
  filters: { search: '', role: 'all' },
  pagination: { page: 1, limit: 10 },
  stats: { total: 0, active: 0 }
};
```

useReducer handles multiple sub-values in a single, predictable way.

7 State Logic

Centralize all state update logic.

Action (What happened) → dispatch() → Reducer (State logic) → New State (Updated)

```
function reducer(state, action) {
  switch (action.type) {
    case 'ADD_USER':
      return {
        ...state,
        users: [...state.users, action.payload]
      };
    case 'REMOVE_USER':
      return {
        ...state,
        users: state.users.filter(
          u => u.id !== action.payload
        )
      };
    default:
      return state;
  }
}
```

All update logic is in one place and easy to manage.

8 Immutable Updates

Always return new state immutably.

Good (Immutable)

```
return {
  ...state,
  count: state.count + 1
};
```

```
return {
  ...state,
  items: [...state.items, newItem]
};
```

Bad (Mutating)

```
state.count += 1;
return state;
```

```
state.items.push(newItem);
return state;
```

Never mutate state. Always return a new object/array. It helps React detect changes correctly.

Pro Tip Use useReducer when state logic becomes complex or depends on previous state.

Best Practice Keep reducer pure, predictable and testable.

Remember Reducer + Dispatch = Powerful combination for complex state.

Custom Hooks

Extract and reuse logic between components.

Extract Logic into Custom Hook → Use in Components → Reuse Logic Anywhere → Clean Code Better Maintainability

What is a Custom Hook?

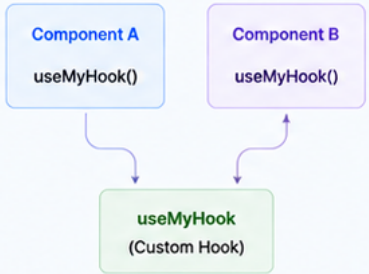
Custom Hooks are JavaScript functions whose names start with "use" and can call other Hooks. They let you extract and reuse logic between components.

```
function useCustomLogic() {
  // use other hooks
  const [state, setState] = useState(null);
  useEffect(() => {
    // logic here
  }, []);

  return { state, setState };
}
```

Custom Hooks follow the same rules as React Hooks.

1 Reuse Logic

Share logic between components.



Write once, use anywhere!

2 useFetch (Data Fetching)

Fetch data with loading, error and data state.

```
import { useState, useEffect } from 'react';
export function useFetch(url) {
  const [data, setData] = useState(null);
  const [loading, setLoading] = useState(true);
  const [error, setError] = useState(null);

  useEffect(() => {
    setLoading(true);
    fetch(url)
      .then(res => res.json())
      .then(json => setData(json))
      .catch(err => setError(err.message))
      .finally(() => setLoading(false));
  }, [url]);

  return { data, loading, error };
}
```

Reusable data fetching logic.

3 useAuth (Auth Logic)

Manage user authentication state.

```
import { useState, useEffect } from 'react';

export function useAuth() {
  const [user, setUser] = useState(null);
  const [loading, setLoading] = useState(true);

  useEffect(() => {
    const savedUser = localStorage.getItem('user');
    if (savedUser) setUser(JSON.parse(savedUser));
    setLoading(false);
  }, []);

  const login = (userData) => {
    localStorage.setItem('user', JSON.stringify(userData));
    setUser(userData);
  };

  const logout = () => {
    localStorage.removeItem('user');
    setUser(null);
  };

  return { user, loading, login, logout };
}
```

Reusable authentication logic.

4 useLocalStorage

Sync state with localStorage.

```
import { useState } from 'react';

export function useLocalStorage(key, initialValue) {
  const [value, setValue] = useState(() => {
    const item = localStorage.getItem(key);
    return item ? JSON.parse(item) : initialValue;
  });

  const setStoredValue = (newValue) => {
    setValue(newValue);
    localStorage.setItem(key, JSON.stringify(newValue));
  };

  return [value, setStoredValue];
}
```

Persistent state across reloads.

5 Hook Rules

Follow these rules in custom hooks.

Only call Hooks at the top level
Don't call Hooks inside loops, conditions or nested functions.

Only call Hooks from React functions
Call Hooks only from React function components or other custom Hooks.

Follow naming convention
Custom Hook names must start with "use".

Following rules keeps your app predictable.

6 Return Value

Return whatever the component needs.

```
export function useToggle(initial = false) {
  const [value, setValue] = useState(initial);
  const toggle = () => setValue(v => !v);

  return {
    value,     // current state
    toggle,    // function to toggle
    setValue   // in case direct update is needed
  };
}
```

Return values can be state, functions or objects.

Destructure in components for clean usage.

7 Usage Example

Use custom hooks in your components.

Component

```
import { useFetch } from './useFetch';
function Users() {
  const { data, loading, error } =
    useFetch('https://api/users');

  if (loading) return <p>Loading...</p>;
  if (error) return <p>Error: {error}</p>;

  return (
    <ul>
      {data.map(u => (
        <li key={u.id}>{u.name}</li>
      ))}
    </ul>
  );
}
```

Custom Hook

```
export function useFetch(url) {
  const [data, setData] = useState(null);
  const [loading, setLoading] = useState(true);
  const [error, setError] = useState(null);

  useEffect(() => {
    // fetch logic...
  }, [url]);

  return { data, loading, error };
}
```

Components stay clean. Logic stays in custom hooks.

8 Benefits

Why use custom hooks?

Code Reusability
Extract logic and reuse across components.

Clean Components
Keep components focused on UI.

Easier Maintenance
Update logic in one place.

Better Readability
Meaningful names make code easier to understand.

Custom Hooks = Clean, Reusable & Scalable Code 🚀

Pro Tip Break complex logic into small custom hooks. Each hook should do one thing well.

Best Practice Name hooks clearly and keep them focused and composable.

Remember Custom hooks use other hooks. Follow the rules and keep it simple.

React Fragment

Group elements without adding extra nodes to the DOM.

Fragment — Group Elements → No Extra Div — Clean DOM → Lightweight — Zero Overhead → Clean & Simple — Better Code

What is React Fragment?

React Fragment lets you group multiple elements in JSX without adding extra nodes to the DOM.

```jsx
// Without Fragment (extra div)
return (
  <div className="wrapper">
    <h1>Hello</h1>
    <p>Welcome to React!</p>
  </div>
);
```

Use Fragment to keep your DOM clean and your code simple.

1 Using <>/> (Short Syntax)

Most common way to use Fragment.

```jsx
function Welcome() {
  return (
    <>
      <h1>Hello!</h1>
      <p>Welcome to React!</p>
      <ul>
        <li>Learn</li>
        <li>Build</li>
        <li>Grow</li>
      </ul>
    </>
  );
}
```

<> </> is a shorthand for <React.Fragment></React.Fragment>

2 Using <React.Fragment>

Full syntax (use when you need key).

```jsx
import React from 'react';

function List() {
  return (
    <React.Fragment>
      <h1>Items</h1>
      {['A', 'B', 'C'].map(item => (
        <p key={item}>{item}</p>
      ))}
    </React.Fragment>
  );;
}
```

Use this syntax when you need to pass a key to the Fragment.

3 Why Use Fragment?

Better structure, cleaner output.

- **No Extra Div** — Prevents unnecessary wrapper elements in the DOM.
- **Clean DOM** — Keeps your HTML structure simple and semantic.
- **Better Performance** — Slightly better rendering and less memory usage.
- **Better Readability** — Code looks clean and easy to understand.

4 Fragment with key

Useful in lists when returning multiple items.

```jsx
function Users() {
  const users = ['John', 'Jane', 'Bob'];

  return users.map(user => (
    <React.Fragment key={user}>
      <h3>{user}</h3>
      <p>Active User</p>
      <hr />
    </React.Fragment>
  ));
}
```

key can only be used with <React.Fragment>, not with <>.

5 Grouping Elements

Return multiple elements as a single unit.

JSX

```jsx
return (
  <>
    <Header />
    <Main />
    <Footer />
  </>
);
```

Output (DOM)

```
<body>
  <header />
  <main />
  <footer />
</body>
```

All elements are grouped without adding extra nodes.

6 Return Multiple

Fragments allow returning multiple siblings.

Without Fragment (Error)

```jsx
return (
  <h1>Hello</h1>
  <p>Welcome</p>
);
```

Error: Adjacent JSX elements must be wrapped in an enclosing tag.

With Fragment (Works)

```jsx
return (
  <>
    <h1>Hello</h1>
    <p>Welcome</p>
  </>
);
```

Fragment fixes the error without extra div.

7 Lightweight & Efficient

Zero impact on DOM and styling.

Fragment keeps DOM shallow and efficient.

8 Best Practices

Use Fragment smartly.

- Use <> </> for most cases.
- Use <React.Fragment> when you need key.
- Use Fragment to keep DOM clean and semantic.
- Makes code easier to read and maintain.

Pro Tip — Prefer Fragment over extra div for better structure and performance.

Best Practice — Use Fragment by default unless you need an extra wrapper.

Remember — Clean DOM leads to better performance, maintainability, and SEO.

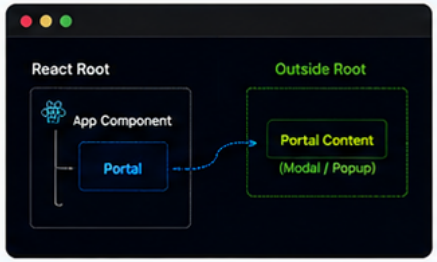
React Portal

Render children into a DOM node **outside** the parent component hierarchy.

Portal — Render Outside Root → Modal / Popup — Show UI on Top → Overlay / Tooltip — Layered UI → ReactDOM.createPortal — Power Behind Portal → Better UI Layer — Clean & Scalable

What is React Portal?

Portals let you render children into a DOM node that exists outside the DOM hierarchy of the parent component.



Great for Modals, Popups, Tooltips, Toasts, and Overlays.

1 Basic Syntax

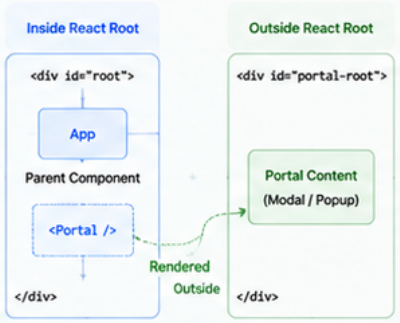
Use ReactDOM.createPortal().

```
import React from 'react';
import ReactDOM from 'react-dom';

function PortalExample() {
  const portalElement = document.getElementById('portal-root');

  return ReactDOM.createPortal(
    <div className="portal-content">
      Hello from Portal! 👋
    </div>,
    portalElement
  );
}
```

First argument: what to render
Second argument: where to render

2 How It Works

Portal renders outside the parent DOM.



Portal keeps the same React behavior (state, context, events). Only the DOM position is outside.

3 Common Use Cases

Perfect for layered UI components.

- **Modal** — Dialogs, forms, confirmations
- **Popup** — Custom popups & alerts
- **Tooltip** — Rich information on hover
- **Overlay** — Background overlay / dimming
- **Toast / Notification** — Top-right messages

Keep UI on top, separate from layout.

4 Example: Modal with Portal

Render a modal outside the root.

```
function Modal({ children, onClose }) {
  const modalRoot = document.getElementById('modal-root');

  if (!modalRoot) return null;

  return ReactDOM.createPortal(
    <div className="overlay" onClick={onClose}>
      <div className="modal" onClick={(e) => e.stopPropagation()}
      >
        {children}
        <button onClick={onClose}>Close</button>
      </div>
    </div>,
    modalRoot
  );
}
```

Rendered into #modal-root
Works like any React component.

5 ReactDOM.createPortal

The API that makes it possible.

```
ReactDOM.createPortal(
  children: ReactNode,
  container: Element,
  key?: string | number
): ReactPortal
```

- children – What you want to render
- container – Target DOM node
- key – (Optional) For reconcilication

Returns a ReactPortal object.

6 Outside Root

Portal content lives outside the root node.

index.html

```
<body>
  <div id="root"></div>
  <!-- React App Root -->

  <div id="portal-root"></div>
  <!-- Portal Target -->

</body>
```

Ensure the target node exists in HTML.

7 UI Layer Benefits

Better control over layering & stacking.

- Always on top
- No z-index issues
- Cleaner structure
- Reusable & scalable
- Better user experience

Build a reliable & consistent UI layer.

8 Best Practices

Use Portals wisely.

- Use portals for UI elements that break out of layout.
- Keep portal root(s) in index.html.
- Handle focus & accessibility in modals.
- Clean up event listeners & side effects.
- Don't overuse portals for simple UI.
- Use when necessary, not everywhere.

Pro Tip Portals don't break React behavior. State, context, and events still work!

Best Practice Use portals for modals, popups, tooltips, toasts, and overlays.

Remember Portal changes only the DOM position, not the React tree.

Component Lifecycle

Understand how a React component mounts, updates, and unmounts using useEffect.

Mount — Component is inserted into the DOM → **Update** — Props or State change causes re-render → **Unmount** — Component is removed from the DOM

Lifecycle
useEffect lets you run side effects at different stages of the component lifecycle.

What is Component Lifecycle?

Every React component has a lifecycle with three main phases.

- **M Mount** — Component is created and inserted into the DOM.
- **U Update** — Component re-renders when props or state change.
- **X Unmount** — Component is removed from the DOM and cleaned up.

useEffect hook helps us perform side effects in these phases.

1 Mount (Initial Render)

Runs when the component is added to the DOM.

```
useEffect(() => {
  console.log('Component Mounted');
  // Side effect (e.g., API call, subscription)

  return () => {
    console.log('Cleanup on Unmount');
  };
}, []);  // Empty dependency -> runs only once
```

Runs once after the first render.
Great for initial data fetch or setup.

2 Update (Re-render)

Runs when dependencies change.

```
const [count, setCount] = useState(0);

useEffect(() => {
  console.log('Count changed:', count);
  // Runs after every update when 'count'
  // value changes
}, [count]);  // Dependency array
```

Runs after re-render when dependencies change.
Keep dependency array accurate.

3 Unmount (Cleanup)

Cleanup function runs before component is removed from the DOM.

```
useEffect(() => {
  const timer = setInterval(() => {
    console.log('Running...');
  }, 1000);

  return () => {
    clearInterval(timer);
    console.log('Component Unmounted');
  };
}, []);
```

Cleanup prevents memory leaks.
Always return a cleanup function for subscriptions, timers, listeners.

useEffect Flow

Here's the complete flow:

Mount
useEffect runs (setup)

↓

Update
Dependencies change
→ cleanup runs (old)
→ useEffect runs (new)

↓

Unmount
Cleanup runs
Component removed

useEffect Syntax

useEffect accepts two arguments.

```
useEffect(() => {
  // Side effect logic

  return () => {
    // Cleanup logic (optional)
  };
}, [dependencies]);
```

- Effect function (runs after render)
- Cleanup function (runs before unmount or before next effect)
- Dependencies array (controls when to run)

Dependency Array Guide

Controls when the effect runs.

[]	Empty Array	Runs once on mount (like componentDidMount)
[dep]	With Dependencies	Runs on mount and when dep changes
----	No Dependencies	Runs after every render (Not recommended)
[A, B]	Multiple Dependencies	Runs when A or B changes

Common Examples

Real-world use cases.

Fetch Data on Mount
```
useEffect(() => { fetchData(); }, []);
```

Update Document Title
```
useEffect(() => { document.title = title; }, [title]);
```

Set Interval
```
useEffect(() => {  const id = setInterval(tick, 1000);
  return () => clearInterval(id); }, []);
```

Add Event Listener
```
useEffect(() => {  window.addEventListener('resize', onResize);
return () => window.removeEventListener('resize', onResize); }, []);
```

Best Practices

- Always provide a dependency array to avoid unnecessary re-renders.
- Always clean up side effects to prevent memory leaks.
- Include all external values in the dependency array.
- Keep effects focused and small.
- Avoid using useEffect for simple state calculations.

Pro Tip — useEffect runs after the component renders and the DOM is updated.

Best Practice — Use the correct dependencies to avoid bugs and infinite loops.

Remember — Mount → Update → Unmount
This is the lifecycle!

useEffect = Side Effect Manager
Powerful but use it wisely.

React Router

Declarative routing for React applications.
Build single page apps with navigation & nested routes.

BrowserRouter → Router Provider
Routes → Route Container
Route → Map Path to UI
Link / NavLink → Navigation
Navigate → Programmatic Nav
Outlet → Render Child
Nested Routes → UI Hierarchy

What is React Router?

React Router is the standard routing library for React. It lets you build SPA navigation with dynamic routes, nested routes, and programmatic navigation.

Enables client-side routing without full page reloads.

1 BrowserRouter

Wrap your app with BrowserRouter.

index.jsx

```jsx
import { BrowserRouter }
from 'react-router-dom';

ReactDOM.createRoot(root).render(
  < BrowserRouter>
    <App />
  </BrowserRouter>
);
```

Uses HTML5 history API to keep UI in sync with URL.

2 Routes

Routes is the container for all Route elements.

App.jsx

```jsx
import { Routes, Route }
from 'react-router-dom';

<Routes>
  <Route path="/" element={<Home />} />
  <Route path="/about" element={<About />} />
</Routes>
```

Matches the current location and renders the best Route.

3 Route

Route maps a path to a UI element.

App.jsx

```jsx
<Route
  path="/users/:id"
  element={<User />}
/>
```

Supports dynamic segments, params, etc.

4 Link

Link is used for navigation without page reload.

Navbar.jsx

```jsx
import { Link }
from 'react-router-dom';

<nav>
  <Link to="/">Home</Link>
  <Link to="/about">About</Link>
  <Link to="/users">Users</Link>
</nav>
```

Renders an accessible anchor (<a>) tag.

5 NavLink

NavLink adds active styles for the current link.

Navbar.jsx

```jsx
import { NavLink }
from 'react-router-dom';

<NavLink
 to="/"
 className={({ isActive }) =>
   isActive ? 'active' : ''
 }
>Home</NavLink>
```

Great for active link highlighting.

6 Navigate

Navigate programmatically change the route.

Login.jsx

```jsx
import { Navigate }
from 'react-router-dom';

if (isLoggedIn) {
  return <Navigate
    to="/dashboard"
    replace
  />;
}
```

Use in events, auth checks, effects, etc.

7 Outlet

Outlet renders child routes in nested routing.

Layout.jsx

```jsx
import { Outlet }
from 'react-router-dom';

export default function
Layout() {
 return (
   <div>
     <header>Header</header>
     <main>
       <Outlet />
     </main>
   </div>
 );
}
```

Placeholder for nested child route elements.

8 Nested Routes

Create UI hierarchies with parent & child routes.

```
/
├── / (Layout)
│   ├── (Home)
│   ├── /about
│   ├── /users (UsersLayout)
│   │   ├── /users (UserList)
│   │   └── /users/:id (UserDetail)
└── └── (NotFound)
```

App.jsx

```jsx
<Routes>
  <Route path="/" element={<Layout />}>
    <Route index element={<Home />} />
    <Route path="about" element={<About />} />
    <Route path="users" element={<UsersLayout />}>
      <Route index element={<UserList />} />
      <Route path=":id" element={<UserDetail />} />
    </Route>
    <Route path="*" element={<NotFound />} />
  </Route>
</Routes>
```

Nested routes + Outlet = Powerful & maintainable UI structure.

Common Hooks & Utilities

useNavigate	Programmatic navigation	`const navigate = useNavigate(); navigate('/dashboard');`
useParams	Access route params	`const { id } = useParams();`
useLocation	Get current location	`const location = useLocation(); console.log(location.pathname);`
useSearchParams	Read / set query params	`const [searchParams] = useSearchParams(); searchParams.get('q');`
useMatch	Check if route matches	`const match = useMatch('/users/:id');`

These hooks help you build dynamic & interactive routing.

Best Practices

- Use BrowserRouter at the root of your app.
- Prefer Link / NavLink over <a> for navigation.
- Use Navigate for redirects (auth, errors, success).
- Use Outlet for clean nested route layouts.
- Keep routes organized and split by feature.
- Handle 404 pages with a wildcard route (*).
- Lazy load routes for better performance.

Pro Tip React Router v6+ uses element instead of component prop.

Best Practice Keep routes simple, clear, and user focused.

Remember URL is the source of truth. Keep UI and URL in sync.

React Router makes your SPA feel native & fast! 🚀

Dynamic Routing

Build powerful, data-driven apps with dynamic routes.
Match URLs, read params, and navigate with ease.

Dynamic Route /:id → URL /blog/123 → Params { id: "123" } → useParams Access in Component → Navigation Link / Navigate → Route Match Component Rendered

What is Dynamic Routing?

Dynamic routing lets you create routes with placeholders (params) that match multiple URLs and show dynamic content.

https://yourapp.com/blog/learn-react

- Create flexible, reusable routes
- Read params using useParams()
- Build nested and data-driven UIs
- Perfect for detail pages, slugs, etc.

1 Dynamic Route

Use :param in route path.

App.jsx

```jsx
import { Routes, Route } from 'react-router-dom';
import BlogPost from './pages/BlogPost';

function App() {
  return (
    <Routes>
      <Route path="/" element={<Home />} />
      <Route path="/blog/:id" element={<BlogPost />} />
      <Route path="/user/:userId" element={<User />} />
    </Routes>
  );
}
```

:id and :userId are dynamic segments (params).

2 Params & useParams

Access route params in your component.

BlogPost.jsx

```jsx
import { useParams } from 'react-router-dom';

export default function BlogPost() {
  const { id } = useParams(); // { id: "123" }

  return (
    <div className="post">
      <h2>Blog Post ID: {id}</h2>
      <p>Load data for post with id: {id}</p>
    </div>
  );
}
```

useParams() returns an object of all matched params.

3 URL & Params

URL changes, params change.

URL	Params (useParams)
/blog/123	{ id: "123" }
/blog/hello-world	{ id: "hello-world" }
/user/john	{ userId: "john" }
/product/45/review/7	{ productId: "45", reviewId: "7" }

Anything in the URL that matches the :param is available in useParams().

4 Slug Example

Use slugs for clean, SEO-friendly URLs.

BlogPost.jsx

```jsx
const { slug } = useParams();

return (
  <div>
    <h2>{slug}</h2>
    <p>Clean URL: /blog/{slug}</p>
  </div>
);
```

Slugs like "learn-react" are easier to read and share.

5 Nested Route

Create child routes inside a parent route.

- /dashboard (Layout)
  - index (Overview)
  - /dashboard/profile
  - /dashboard/settings

App.jsx

```jsx
<Route path="/dashboard" element={<DashboardLayout />}>
  <Route index element={<Overview />} />
  <Route path="profile" element={<Profile />} />
  <Route path="settings" element={<Settings />} />
</Route>
```

Child routes render inside the parent using <Outlet />.

6 Nested Route with Outlet

Parent layout renders children via <Outlet />.

DashboardLayout.jsx

```jsx
import { Outlet, Link } from 'react-router-dom';

export default function DashboardLayout() {
  return (
    <div className="dashboard">
      <aside>
        <Link to="/dashboard">Overview</Link>
        <Link to="/dashboard/profile">Profile</Link>
        <Link to="/dashboard/settings">Settings</Link>
      </aside>
      <main>
        <Outlet /> {/* Child route will render here */}
      </main>
    </div>
  );
}
```

<Outlet /> is a placeholder for nested route content.

7 Route Match

Route matches when the URL fits the path.

Route Path	URL	Match?
/blog/:id	/blog/123	Yes
/blog/:id	/blog/abc	Yes
/blog/:id	/blog	No
/user/:userId	/user/john	Yes
/user/:userId	/user/	No

Exact segments must match. Params can be anything.

8 Navigation

Navigate to dynamic routes easily.

Navigation.jsx

```jsx
import { Link, useNavigate } from 'react-router-dom';

export default function Navigation() {
  const navigate = useNavigate();
  const postId = 42;

  return (
    <div>
      <Link to={`/blog/${postId}`}>View Post</Link>
      <button onClick={() => navigate(`/blog/${postId}`)}>
        Go to Post
      </button>
    </div>
  );
}
```

Use template literals or navigate() for dynamic paths.

Common Patterns

Multiple Params

```
/product/:category/:id
```

```jsx
const { category, id } = useParams();
```

Optional Param (v6.4+)

```
/search/:query?
```

```jsx
const { query } = useParams();
```

Wildcard (Not Found)

```
/docs/*
```

```jsx
const params = useParams();
```

Best Practices

- Use meaningful slugs (e.g., /blog/learn-react).
- Keep routes simple and focused.
- Always validate params before using.
- Use nested routes & Outlet for clean layouts.
- Prefer Link over <a> for navigation.

Pro Tip

Combine dynamic routes with data fetching (like APIs) to build detail pages that load content based on the URL params.

```jsx
const { id } = useParams();
useEffect(() => { fetch(`/api/post/${id}`); }, [id]);
```

Quick Reference

:param	Dynamic segment
useParams()	Read params in component
<Link to="...">	Navigate declaratively
<Outlet />	Render nested child routes
useNavigate()	Imperative navigation

Navigation Hooks

Powerful hooks to navigate programmatically and control browser history in React Router.

useNavigate Navigation Hook → Navigate Go to a Route → Back Go to Previous → Forward Go to Next → Replace Replace Current → History Browser History → Redirect To Another Route → State Pass Data Along

What is Navigation Hook?

React Router provides `useNavigate()` hook to navigate programmatically without using `<Link>`.

- Imperative navigation
- Control browser history
- Pass state between routes
- Replace, go back, forward
- Redirect & conditional navigation

1 useNavigate Hook

Use useNavigate() to get the navigate function.

Example

```
import { useNavigate } from 'react-router-dom';

function MyComponent() {
  const navigate = useNavigate();
  return (
    <button onClick={() => navigate('/dashboard')}>
      Go to Dashboard
    </button>
  );
}
```

`const navigate = useNavigate();`

2 Navigate to a Route

Navigate to different routes.

Example

```
navigate('/home');     // basic navigation
navigate('/about', { replace: true });
navigate('/profile', {
  state: { from: 'dashboard' }
});
```

`navigate(path, options?)`

3 Go Back / Forward

Navigate through history stack.

Example

```
navigate(-1);     // back
navigate(1);      // forward
navigate(-2);     // go back 2 steps
```

`-1 = back, 1 = forward, -n = back n steps`

4 Replace Current Entry

Replace current history entry instead of pushing a new one.

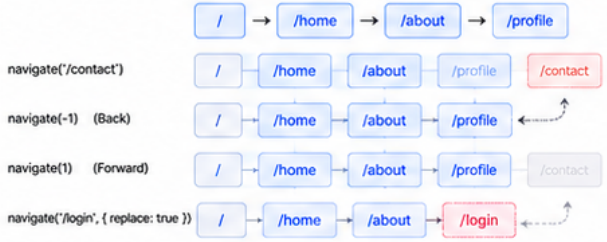
Example

```
navigate('/login', { replace: true });
navigate(-1, { replace: true });
```

Does not add a new entry in the history stack.

5 History Stack (Visual)

Understand how navigation affects history.


/ → /home → /about → /profile
navigate('/contact') / /home /about /profile /contact
navigate(-1) (Back) / /home /about /profile
navigate(1) (Forward) / /home /about /profile /contact
navigate('/login', { replace: true }) / /home /about /login

Navigation manipulates the browser history stack.

6 Redirect (Conditional)

Redirect users based on conditions.

Example

```
const isAuth = false;

if (!isAuth) {
  navigate('/login', {
    state: { from: '/dashboard' },
    replace: true
  });
}
```

Great for auth checks, 404, or role-based access.

7 Pass State Between Routes

Send data to the next route.

Example

```
navigate('/user/123', {
  state: {
    user: { id: 123, name: 'John' }
  }
});
```

Access with `useLocation().state` in the target component.

8 Access State in Target

Read state in the destination component.

Example

```
import { useLocation } from 'react-router-dom';

function User() {
  const location = useLocation();
  const state = location.state;
  console.log(state?.user);
  return <div>User Details</div>;
}
```

`location.state` contains the passed data.

9 Quick Reference

`navigate('/path')`	Go to a path
`navigate(-1)`	Go back
`navigate(1)`	Go forward
`navigate('/path', { replace: true })`	Replace current entry
`navigate('/path', { state })`	Pass state to route
`useNavigate()`	Get navigate function
`useLocation().state`	Access passed state

Pro Tip

Use `navigate()` for buttons, forms, timers, and conditional logic.

Best Practices

- ✓ Use Link/NavLink for normal navigation.
- ✓ Use navigate for programmatic navigation.
- ✓ Use `replace: true` for redirects.
- ✓ Pass meaningful state, not large objects.

Common Use Cases

- Login redirects
- Role-based navigation
- Form submission
- Multi-step flows
- Go back after actions
- Wizards / Steppers
- Handle 404 / errors
- Conditional routing

One Hook, Many Powers!

```
const navigate = useNavigate();
navigate('/dashboard', { state: { from: '/home' } });
navigate(-1);  // back
navigate('/login', { replace: true });
```

API Fetching in React

Communicate with APIs and handle data using fetch, Axios, and Async/Await.

Request — Send API Request → API Server — Process Request → JSON Response — Return Data → Parse & Handle — Use in Component → Render UI — Show Data on Screen

What is API Fetching?

API fetching allows your app to send requests to a server, get data, and update the UI dynamically.

- GET – Read / Fetch Data
- POST – Create Data
- PUT – Update Data
- DELETE – Remove Data
- { } JSON – Data Format

1 Using fetch()

Built-in browser API.

```
async function getUsers() {
  try {
    const res = await fetch('https://
      jsonplaceholder.typicode.com/users');
    const data = await res.json();
    console.log(data);
  } catch (error) {
    console.error('Error:', error);
  }
}
getUsers();
```

fetch() returns a Response object. Use .json() to parse JSON data.

2 Using Axios

Popular promise-based HTTP client.

```
import axios from 'axios';

async function getUsers() {
  try {
    const res = await axios.get(
      'https://jsonplaceholder.typicode.com/users'
    );
    console.log(res.data);
  } catch (error) {
    console.error('Error:', error);
  }
}
getUsers();
```

Axios automatically parses JSON. Data is in res.data

3 GET Request

Fetch data from server.

```
// fetch
const res = await fetch('/api/users');
const users = await res.json();

// axios
const res = await axios.get('/api/users');
const users = res.data;
```

GET — Use GET to retrieve data. Safe & idempotent.

4 POST Request

Send data to create a resource.

```
// fetch
const res = await fetch('/api/users', {
  method: 'POST',
  headers: { 'Content-Type': 'application/json' },
  body: JSON.stringfiy({ name: 'John', age: 25 })
});
const data = await res.json();

// axios
const res = await axios.post('/api/users',
  { name: 'John', age: 25 }
);
```

POST — Use POST to send data. Returns created resource.

JSON Example

```
{
  "id": 1,
  "name": "John Doe",
  "email": "john@example.com",
  "age": 25,
  "isActive": true
}
```

JSON (JavaScript Object Notation) is the most common format for API responses.

5 PUT Request

Update an existing resource.

```
// fetch
const res = await fetch('/api/users/1', {
  method: 'PUT',
  headers: { 'Content-Type': 'application/json' },
  body: JSON.stringfiy({ name: 'John Updated' })
});
const data = await res.json();

// axios
const res = await axios.put('/api/users/1',
  { name: 'John Updated' }
);
```

PUT — Use PUT to update completely an existing resource.

6 DELETE Request

Remove a resource.

```
// fetch
const res = await fetch('/api/users/1', {
  method: 'DELETE'
});

// axios
const res = await axios.delete('/api/users/1');
```

DELETE — Use DELETE to remove a resource.

7 Async / Await

Write async code in clean and readable way.

```
async function loadData() {
  try {
    const res = await fetch('/api/data');
    const data = await res.json();
    setData(data);
  } catch (error) {
    setError(error.message);
  } finaly {
    setLoading(false);
  }
}
```

</> — Async/Await makes code easier to read & maintain.

8 Example: Full CRUD with Axios

Complete example using Axios.

```
import axios from 'axios';

// GET
const getUsers = () => axios.get('/api/users');
// POST
const createUser = (user) => axios.post('/api/users', user);
// PUT
const updateUser = (id, user) => axios.put(`/api/users/${id}`, user);
// DELETE
const deleteUser = (id) => axios.delete('/api/users/${id}');
```

Axios simplifies HTTP requests and error handling.

Best Practices

- Use Async/Await for clean code.
- Always handle errors with try/catch.
- Show loading state while fetching.
- Keep API base URL in config.
- Use environment variables for sensitive data.
- Use proper HTTP methods.
- Validate and sanitize data.

Pro Tip

Use Axios interceptors to add tokens, handle errors globally, and log requests.

Common Status Codes

200 OK – Success	400 Bad Request – Invalid Request
201 Created – Resource Created	401 Unauthorized – Auth Required
	404 Not Found – Resource Not Found
	500 Server Error – Server Problem

Useful Headers

```
Content-Type: application/json
Authorization: Bearer <token>
Accept: application/json
```

Quick Reference

GET	/api/items	→ Read Data	PUT	/api/items/1	→ Update Data
POST	/api/items	→ Create Data	DELETE	/api/items/1	→ Delete Data

Loading & Error Handling

Build resilient React apps with smooth loading states and graceful error handling.

Loading (Show Loader) → Success (Show Data) → Error (Show Error UI) → Retry (Try Again) → Fallback UI (Safe Fallback)

What is Loading & Error Handling?

It ensures your app handles API delays, failures, and unexpected issues gracefully with good user experience.

- Loading – Data is being fetched
- Skeleton – Placeholder UI
- Error – Something went wrong
- Retry – Try the action again
- Fallback UI – Safe fallback content
- Try Catch – Handle exceptions
- Status – HTTP response status

1 Loading State

Show a loader while data is being fetched.

```
const [loading, setLeading] = useState(true);

useEffect(() => {
  fetchData().finally(() => setLaading(false));
}, []);

if (loading) {
  return <div className="center">
    <Spinner />
  </div>;
}
```

Use loading state to inform users that something is in progress.

2 Spinner

Use a spinner for short loading operations.

Loading...

Best for actions that take a few seconds (e.g., button click, data fetch).

3 Skeleton

Use skeleton screens for better UX on longer loads.

Skeleton screens reduce perceived loading time and improve UX.

4 Error State

Show an error message when something fails.

```
if (error) {
  return (
    <div className="error-box">
      <h3>⚠ Oops! Something went wrong.</h3>
      <p>{error.message}</p>
      <button onClick={retry}>Retry</button>
    </div>
  );
}
```

Always show meaningful error messages and a way to recover.

5 Retry

Allow users to retry the failed action.

```
const retry = () => {
  setError(null);
  setLoading(true);
  fetchData();
};
```

Retry

Retry gives control back to the user and improves experience.

6 Fallback UI

Show a fallback UI when data cannot be loaded.

No Data Available

We couldn't load the data right now.

Fallback UI keeps your app stable even when data fails to load.

7 Try Catch (Async)

Use try....catch to handle errors in async calls.

```
const fetchData = async () => {
  setLoading(true);
  setError(null);
  try {
    const res = await fetch('/api/data');
    if (!res.ok) throw new Error('Failed');
    const data = await res.json();
    setData(data);
  } catch (err) {
    setError(err);
  } finally {
    setLoading(false);
  }
};
```

Always wrap async calls in try....catch to handle unexpected errors.

8 Status Codes

Common HTTP status codes to handle.

Code	Meaning	When to Handle
200	OK	Success
201	Created	Resource created
400	Bad Request	Invalid input
401	Unauthorized	Login required
403	Forbidden	No permission
404	Not Found	Resource not found
500	Server Error	Server issue

Use status codes to show right message and take correct action.

9 Complete Example

Putting it all together.

```
const [data, setData] = useState(null);
const [loading, setLoading] = useState(false);
const [error, setError] = useState(null);

const loadData = async () => {
  setLoading(true);
  setError(null);
  try {
    const res = await fetch('/api/data');
    if (!res.ok) throw new Error('Failed');
    setData(await res.json());
  } catch (err) {
    setError(err.message);
  } finally {
    setLoading(false);
  }
};

// Render Logic
if (loading) return <Spinner />;
if (error) return <ErrorUI retry={loadData} />;
if (!data) return <FallbackUI />;
return <DataView data={data} />;
```

Pro Tip

- Always handle loading & error states.
- Don't leave users guessing.
- Good UX = clear feedback + control.

Best Practices

- Use skeleton for page-level loading.
- Show spinner for small operations.
- Provide retry and fallback options.
- Log errors (console or monitoring tool).

Common Patterns

```
if (loading) return <Spinner />;
if (error) return <ErrorUI />;
if (!data) return <FallbackUI />;
return <Content data={data} />;
```

Useful Libraries

- react-hot-toast – Toast notifications
- react-error-boundary – Error boundaries
- swr / react-query – Data fetching
- axios – HTTP client

Remember

Handle every state:

Loading → Success → Error

That's a reliable app! 🚀

React Styling

Style your React apps in multiple ways.
Choose the best approach for your project.

CSS — Global & Classic | CSS Module — Scoped Locally | Inline Style — JS Object Styles | Styled Components — CSS in JS | Tailwind CSS — Utility First CSS | Bootstrap — Powerful Components | Emotion — Performant CSS in JS | CSS with Superpowers

About React Styling

React gives you the freedom to style components in many different ways. Pick the one that fits your project size, team workflow, and performance needs.

- Scalable
- Maintainable
- Performant
- Developer Friendly
- Component Scoped

Tip: You can even use multiple styling approaches together!

1 CSS

Traditional global CSS file.

index.css

```css
.card {
  background: #f0f0f0;
  padding: 16px;
  border-radius: 8px;
}

title {
  color: #1976d2;
  font-size: 20px;
  font-weight: 600;
}
```

Card.jsx

```jsx
import './index.css';

export default function
Card() {
  return (
    <div className="card">
      <h2 className="title">
        Hello React
      </h2>
    </div>
  );
}
```

Pros: Simple, familiar, works everywhere.
Cons: Global scope, class name conflicts.

2 CSS Module

Scoped CSS by default.

Card.module.css

```css
.card {
  background: #e3f2fd;
  padding: 16px;
  border-radius: 8px;
}

title {
  color: #1565c0;
  font-weight: 600;
}
```

Card.jsx

```jsx
import styles from
'./Card.module.css';

export default function
Card() {
  return (
    <div className={styles.card
      <h2 className={styles.titl
        Hello React
      </h2>
    </div>
  );
}
```

Pros: Scoped, prevents conflicts.
Cons: Need to import for each file.

3 Inline Style

Use JS objects for styling.

Card.jsx

```jsx
const cardStyle = {
  background: '#fff3e0',
  padding: '16px',
  borderRadius: '8px',
};

const titleStyle = {
  color: '#e65100',
  fontSize: '20px',
  fontWeight: 600,
};

export default function Card() {
  return (
    <div style={cardStyle}>
      <h2 style={titleStyle}>Hello React</h2>
    </div>
  );
}
```

Pros: Dynamic styles, no extra files.
Cons: No pseudo-classes, hard to maintain.

4 Styled Components

CSS-in-JS with tagged template literals.

Card.jsx

```jsx
import styled from 'styled-components';

const CardWrapper = styled.div`
  background: #f3e5f5;
  padding: 16px;
  border-radius: 8px;
`;

const Title = styled.h2`
  color: #7b1fa2;
  font-size: 20px;
  font-weight: 600;
`;

export default () => (
  <CardWrapper>
    <Title>Hello React</Title>
  </CardWrapper>
);
```

Pros: Scoped, dynamic, powerful.
Cons: Extra library, bigger bundle.

5 Tailwind CSS

Utility-first CSS framework.

Card.jsx

```jsx
export default function Card() {
  return (
    <div className="bg-blue-50 p-4 rounded-xl shadow-sm">
      <h2 className="text-blue-700 text-xl font-semibold">
        Hello React
      </h2>
    </div>
  );
}
```

Pros: Fast, consistent, responsive, low CSS.
Cons: Learning curve, class names in JSX.

6 Bootstrap

Responsive UI components & utilities.

Card.jsx

```jsx
export default function Card() {
  return (
    <div className="card shadow-sm p-3">
      <h2 className="card-title text-primary mb-0">
        Hello React
      </h2>
    </div>
  );
}
```

Pros: Ready components, grid, responsive.
Cons: Larger CSS, less customization.

7 Emotion

Performant CSS-in-JS library.

Card.jsx

```jsx
/** @jsxImportSource @emotion/react */
import { css } from '@emotion/react';

const card = css`
  background: #fff0f6;
  padding: 16px;
  border-radius: 8px;
`;
const title = css`
  color: #c2185b;
  font-size: 20px;
  font-weight: 600;
`;
export default () => ( <div css={card}>
                         <h2 css={title}>Hello React</h2>
```

Pros: Fast, flexible, zero-runtime cost.
Cons: JS knowledge needed.

8 SCSS

CSS with variables, nesting, mixins.

styles.scss

```scss
$primary: #1976d2;

card {
  background: lighten($primary, 40%);
  padding: 16px;
  border-radius: 8px;

  .title {
    color: $primary;
    font-weight: 600;
  }
}
```

Card.jsx

```jsx
import './styles.scss';

export default function Card() {
  return (
    <div className="card">
      <h2 className="title">
        Hello React
      </h2>
    </div>
  );
}
```

Pros: Powerful CSS features, reusable.
Cons: Needs build setup (Sass compiler).

When to Use What?

- CSS / SCSS → Small projects, traditional way
- CSS Module → Scoped styles without extra libs
- Inline Style → Dynamic styles, quick & simple
- Styled Components / Emotion → CSS-in-JS lovers
- Tailwind CSS → Rapid UI with utility classes
- Bootstrap → Need ready components fast

Quick Comparison

Approach	Scoped	Dynamic	Performance	Bundle Size	Learning Curve
CSS	✗	⚠	★	Small	Easy
CSS Module	✓	⚠	★★	Small	Easy
Inline Style	✗	✓	★	Small	Easy
Styled Components	✓	✓	★★	Medium	Medium
Tailwind CSS	✓	✓	★★★	Small	Medium
Bootstrap	✗	⚠	★	Large	Easy
Emotion	✓	✓	★★★	Medium	Medium
SCSS	✗	⚠	★★	Small	Medium

Best Practices

- Keep styles modular and reusable.
- Avoid !important.
- Follow naming conventions (BEM, etc.).
- Use variables for colors & spacing.
- Prefer responsive & accessible styles.

Pro Tip

You can combine approaches!
Example: Tailwind + CSS Module
or Styled Components + Global CSS.

```jsx
<div className="p-4">
  <StyledButton>Click</StyledButton>
</div>
```

Images & Assets in React

Manage and use images, icons, SVGs, logos, backgrounds and other static assets the right way.

Import Image – Use in Components | SVG – Scalable Vector | Icons – Icon Libraries | Public Folder – Static Files | Assets – Organized Files | Logo – Brand Identity | Background – Set Backgrounds | Optimization – Fast & Optimized

About Images & Assets

React apps use different ways to include images and assets. Choose the best approach based on type and use case.

- Faster Loading
- Better Performance
- Scalable & Responsive
- SEO Friendly
- Organized Structure

Tip: Use the right format and size for the best UX.

1 Import Image (Recommended)

Import images inside the src folder.

```
src/
  assets/
    logo.png
    banner.jpg
    avatar.svg
  components/
    Header.jsx
```

Header.jsx

```jsx
import logo from
  '../assets/logo.png';

function Header() {
  return (
    <header>
      <img src={logo}
           alt="Logo"
           width={120} />
    </header>
  );
}
```

Pros: Bundled, optimized, hashed filenames.
Cons: Not for large or dynamic files.

2 SVG (Scalable Vector Graphics)

Perfect for logos, icons and illustrations.

icon.svg

```svg
<svg width="24" height="24"
  viewBox="0 0 24 24"
  fill="currentColor"
  xmlns="http://www.w3.org/2000/svg">
    <path d="M12 219 19H3L12 2z"/>
</svg>
```

Component.jsx

```jsx
import { ReactComponent as Icon }
from '../assets/icon.svg';
<Icon width={24} height={24} />
```

Pros: Scalable, small size, customizable.
Cons: Not ideal for photos.

3 Icons (Icon Libraries)

Use icon libraries for ready-made icons.

Popular Libraries

- react-icons
- @heroicons/react
- lucide-react
- fontawesome

Example (react-icons)

```jsx
import { FaUser, FaHome }
  from 'react-icons/fa';

function Navbar() {
  return (
    <nav>
      <FaHome />
      <FaUser />
    </nav>
  );
}
```

Pros: Easy to use, consistent, customizable.
Cons: Can increase bundle size.

4 Public Folder (Static Files)

Files in public/ are served from the root.

```
public/
  favicon.ico
  robots.txt
  logo192.png
  images/
    hero.jpg
src/
  App.jsx
```

Component.jsx

```jsx
function App() {
  return (
    <img
      src="/images/hero.jpg"
      alt="Hero"
      width={600}
    />
  );
}
```

Pros: Not bundled, good for large/static files.
Cons: No hashing, not processed by Webpack.

5 Assets Organization

Keep assets organized for scalability.

```
src/
  assets/
    images/
      banner.jpg
      background.png
    icons/
      logo.svg
      menu.svg
  components/
```

Best Practices

- Use meaningful names
- Group by type
- Use lowercase & kebab-case
- Keep unused assets clean
- Maintain consistent formats

Pro Tip: A good structure saves time and avoids confusion.

6 Logo Usage

Use logos in header, footer, auth, etc.

logo.png

Header.jsx

```jsx
import logo from
  '../assets/logo.png';

<header className="header">
  <img src={logo} alt="MyBrand"
       className="logo" />
</header>
```

Tip: Use SVG logos for better quality on all screens.

7 Background Images

Set background images in components.

CSS (Recommended)

```css
.hero {
  background-image: url('/images/
  hero-bg.jpg');
  background-size: cover;
  background-position: center;
  height: 100vh;
}
```

Inline Style

```jsx
const style = {
  backgroundImage:
   "url(/images/hero-bg.jpg)",
  backgroundSize: 'cover',
  backgroundPosition: 'center',
  height: '100vh',
};
<section style={style} />
```

Tip: Use optimized images for backgrounds.

8 Optimization Best Practices

Optimize assets for speed & performance.

- Use proper image formats (WebP, AVIF when possible)
- Compress images (TinyPNG, ImageOptim)
- Use responsive images (srcSet, sizes)
- Lazy load images (loading="lazy")
- Avoid large images in the bundle

Example (Lazy Load)

```jsx
<img
  src={image}
  alt="Lazy"
  loading="lazy"
  width="600"
  height="400"
/>
```

Tip: Smaller assets = Faster apps = Happy users!

Image Formats Guide

- JPEG – Best for photos
- PNG – Best for graphics with transparency
- SVG – Best for icons & logos
- WebP – Best for performance (smaller size)
- AVIF – Next-gen format (best compression)

Quick Comparison

Approach	Scalable	Customizable	Performance	Best For
Import Image	✓	⚠	★★☆	General Images
SVG	✓	✓	★★★★	Logos, Icons
Icon Libraries	✓	⚠	★★★☆	UI Icons
Public Folder	✗	✗	★★★☆	Large Static Files
Background Images	✓	✓	★★★☆	Sections, Hero

Useful Libraries

- react-icons – Popular icon library
- @heroicons/react – Beautiful SVG icons
- lucide-react – Lightweight icons
- @svgr/webpack – Import SVG as components
- image-size – Get image dimensions

Pro Tip

Always optimize your images and assets.
Better performance = Better rankings & UX.

Choose Right Format → Compress Image → Optimize Delivery → Better Performance

Forms Validation in React

Validate user input, show meaningful error messages and submit forms with confidence.

Validation	Required	Regex	Formik	React Hook Form	Yup	Error Message	Submit
Ensure Correct Input	Mark Fields Required	Pattern Matching	Form Management	Performant & Minimal	Schema Validation	Show Helpful Errors	Handle Form Submit

What is Forms Validation?

Form validation ensures that the data entered by users is correct, complete, and safe before submitting.

- Improve user experience
- Prevent invalid submissions
- Show helpful error messages
- Ensure data integrity
- Follow rules and patterns

Validate early, show clear errors, and make it easy to submit.

1 Validation Basics

Basic validation using state and conditions.

```
const [form, setForm] = useState({
  name: '',
  email: '',
  age: ''
});
const [errors, setErrors] = useState({});

const validate = () => {
  const newErrors = {};
  if (!form.name) newErrors.name = 'Name is required';
  if (!form.email) newErrors.email = 'Email is required';
  setErrors(newErrors);
  return Object.keys(newErrors).length === 0;
};
```

Simple, flexible and easy to understand.

2 Required Field

Make a field mandatory.

```
{!form.name && (
  <p className="error">
    Name is required
  </p>
)}
```

Name *

Enter your name

Name is required

Use * and clear messages for required fields.

3 Regex Validation

Validate using regular expressions.

```
const emailRegex = /^[^\s@]+@[^\s@]+\.[^\s@]+$/;

const validateEmail = (email) => {
  return emailRegex.test(email);
};
const mobileRegex = /^[6-9]\d{9}$/;

const validateMobile = (mobile) => {
  return mobileRegex.test(mobile);
};
```

Pattern	Use Case
emailRegex	Email format
mobileRegex	Mobile number
alphaRegex	Only letters

Regex helps to match specific patterns.

4 Error Message

Show error messages below fields.

Email *

abc@gmail

Please enter a valid email address.

Password *

123

Password must be at least 6 characters.

Show specific and friendly error messages.

5 Formik

Popular form library for React.

Install

```
npm install formik yup
```

Example

```
import { Formik, Form, Field, ErrorMessage }
    from 'formik';
import * as Yup from 'yup';

const schema = Yup.object({
  name: Yup.string().required('Required'),
  email: Yup.string().email('Invalid email')
      .required('Required'),
});

<Formik
  initialValues={{ name: '', email: '' }}
  validationSchema={schema}
  onSubmit={(values) => console.log(values)}
>
  <Form>
    <Field name="name" placeholder="Name" />
    <ErrorMessage name="name" component="div" className="error" />
  </Form>
</Formik>
```

Features

- Handles values & errors
- Schema validation with Yup
- Touched, dirty tracking
- Easy error handling

Best For

- Medium to large forms
- Complex validation

Formik + Yup = Powerful Combination

6 React Hook Form

Performant and flexible form library.

Install

```
npm install react-hook-form
```

Example

```
import { useForm } from 'react-hook-form';

const { register, handleSubmit, formState: { errors } } = useForm();

const onSubmit = (data) => console.log(data);

return (
  <form onSubmit={handleSubmit(onSubmit)}>
    <input {...register('name', { required: 'Name is required' })} />
    {errors.name && <p className="error">{errors.name.message}</p>}
    <button type="submit">Submit</button>
  </form>
);
```

Features

- Uncontrolled inputs
- Better performance
- Less re-renders
- Easy to use
- Built-in validation

Best For

- Large forms
- Performance critical
- Minimal re-renders

React Hook Form = Fast, Small, Powerful

7 Yup (Schema Validation)

Schema bulider for value parsing & validation.

```
import * as Yup from 'yup';

const schema = Yup.object({
  name: Yup.string()
    .min(3, 'Min 3 characters')
    .required('Required'),
  email: Yup.string()
    .email('Invalid email')
    .required('Required'),
  password: Yup.string()
    .min(6, 'Min 6 characters')
    .matches(/^(?=.*[A-Z])(?=.*\d).+$/,
      'Must contain 1 uppercase & 1 number')
    .required('Required')
});
```

Yup makes complex validation easy and reusable.

8 Submit Handler

Handle form submit.

```
const handleSubmit = (e) => {
  e.preventDefault();
  if (validate()) {
    // Submit data to API
    console.log('Form submitted', form);
  } else {
    console.log('Fix the errors first');
  }
};
```

Submit

Always validate before submit.

Validation Rules Quick Guide

Rule	Example
Required	!value
Email	/^[^\s@]+@[^\s@]+\.[^\s@]+$/
Mobile (10 digit)	/^[6-9]\d{9}$/
Min Length	value.length >= 6
Alphabets only	/^[A-Za-z]+$/
Alphanumeric	/^[a-zA-Z0-9]+$/

Example Form UI

Name * John Doe

Email * john@gmail.com

Password * •••••••

Submit

Common Error Messages

- This field is required.
- Please enter a valid email.
- Password must be at least 6 characters.
- Must contain 1 uppercase and 1 number.
- Only letters are allowed.
- Please enter a valid phone number.

Best Practices

- Validate on change or on blur.
- Show errors only when needed.
- Use clear and specific messages.
- Group related validations.
- Keep validation logic separate.
- Test all edge cases.

Pro Tip

Use libraries for complex forms, but understand the basics of validation. Great UX = Happy Users!

Validate → Show Errors → Fix Errors → Submit

React Icons

Easily include popular icons in your React apps using react-icons with support for multiple icon libraries.

- **react-icons** — All in one icon library for React
- **Font Awesome** — Access to Font Awesome icons
- **Material Icons** — Material Design icons
- **Heroicons** — Beautiful hand-crafted SVG icons
- **SVG** — Scalable Vector Graphics
- **Import** — Import only what you use
- **Size** — Control icon dimensions
- **Color** — Customize icon colors

1 react-icons

The core package that gives you access to multiple icon libraries in one place.

```
npm install react-icons
# or
yarn add react-icons
```

Huge collection of popular icons from multiple libraries.

2 Font Awesome

Use Font Awesome icons.

```
import { FaUser, FaHome } from 'react-icons/fa';

<FaUser size={20} color="blue" />
<FaHome size={24} color="green" />
```

Examples

FaUser

FaHome

3 Material Icons

Use Material Design icons.

```
import { MdEmail, MdSettings } from 'react-icons/md';

<MdEmail size={24} color="red" />
<MdSettings size={28} color="gray" />
```

Examples

MdEmail

MdSettings

4 Heroicons

Use beautiful hand-crafted SVG icons.

```
import { HomeIcon, UserIcon } from '@heroicons/react/24/outline';

<HomeIcon className="w-6 h-6 text-indigo-600" />
<UserIcon className="w-6 h-6 text-indigo-600" />
```

Examples

HomeIcon

UserIcon

5 SVG

You can also use custom SVG icons.

```
import { ReactComponent as Logo } from './logo.svg';

<Logo width={40} height={40} fill="currettColor" />
```

Examples

Custom SVG

Custom SVG

6 Import

Import only the icons you need.

```
import { FaBeer } from 'react-icons/fa';

<FaBeer />
```

This helps reduce your bundle size and improves performance.

7 Size

Control the size of icons easily.

```
<FaCog size={16} />     {/* small */}
<FaCog size={24} />     {/* medium */}
<FaCog size={32} />     {/* large */}
<FaCog size="2em" />    {/* using em */}
```

Examples

16px

24px

32px

8 Color

Customize icon colors.

```
<FaHeart color="red" />
<MdEmail color="#4CAF50" />
<HomeIcon className="w-6 h-6 text-purple-600" />
```

Examples

Red

Green

Purple

Quick Reference

- Install: npm install react-icons
- Import from library
- Use icon component
- Adjust size with size prop
- Change color with color prop

Common Libraries

Font Awesome	fa
Material Icons	md
Heroicons	heroicons

Example Usage

```
import { FaRocket } from 'react-icons/fa';

function LaunchButton() {
  return (
    <button className="btn btn-primary">
      <FaRocket size={20} color="white" /> Launch
    </button>
  );
}
```

Best Practices

- Import only the icons you use
- Use consistent sizes
- Use CSS / props for color
- Prefer SVG icons for crisp look
- Keep your app bundle small

More Icons

Explore all available icons and libraries at react-icons.github.io

React.memo

Memoize React components to prevent unnecessary re-renders and boost performance.

- **React.memo** — Higher Order Compenent
- **Memoization** — Cache Result & Reuse
- **Prevent Re-render** — Skip Unnecessary Updates
- **Props Comparison** — Shallow Compare By Default
- **Performance** — Better Speed & Efficiency
- **Pure Component** — Like PureComponent for Function
- **Optimization** — Optimize Expensive Components
- **Reusable** — Reusable in Any Project

1 What is React.memo?

React.memo is a higher-order component that memoizes functional components. It prevents re-rendering if props haven't changed (shallow comparison).

Use it to optimize components that re-render frequently with the same props.

2 Basic Usage

```jsx
import React, { memo } from 'react';

const UserCard = memo(function UserCard({ name, age }) {
  console.log('Rendering UserCard');
  return (
    <div className="user-card">
      <h3>{name}</h3>
      <p>Age: {age}</p>
    </div>
  );
});
```

UserCard will re-render only if `name` or `age` changes.

3 How it Works

Parent Re-renders
State or props changed

With React.memo
Child re-renders only if props changed

- **Shallow Comparison** — Compares previous & next props (shallow).
- **Same Props** — If props are same, skip re-render.
- **Different Props** — If props changed, component re-renders.

4 Props Comparison (Default)

```jsx
// Default: Shallow comparison
memo(Component, areEqual?)

// areEqual(prevProps, nextProps)
// return true  -> Skip re-render
// return false -> Re-render
```

By default, React.memo uses `Object.is` for shallow comparison of props.

5 Custom Comparison

```jsx
const areEqual = (prevProps, nextProps) => {
  return prevProps.id === nextProps.id &&
         prevProps.data === nextProps.data;
};

const MyComponent = memo(MyComponent, areEqual);
```

Use custom comparison for deep objects or complex props.

6 Example: Without vs With React.memo

Without React.memo

```jsx
const Child = ({ value }) => {
  console.log('Child rendered');
  return <div>{value}</div>;
};
```

- ✗ Parent re-render
- ✗ Child re-renders every time
- ✗ Poor performance

With React.memo

```jsx
const Child = memo(({ value }) => {
  console.log('Child rendered');
  return <div>{value}</div>;
});
```

- ✓ Parent re-render
- ✓ Child re-renders only if `value` changes
- ✓ Better performance

7 When to Use

- Expensive components
- Large lists
- Pure UI components
- Stable props
- Frequent parent re-renders

Don't use everywhere. Measure performance first!

8 Best Practices

- Use for pure functional components
- Avoid if props are always changing
- Combine with `useCallback` & `useMemo`
- Keep props as primitive values
- Profile before & after optimization

Quick Reference

React.memo(Component)	Memoize component
Shallow Compare	Default comparison
Skip Re-render	If props are same
Re-render	If props are different
Custom Compare	Provide areEqual function

Practical Example

```jsx
const ExpensiveComponent = memo(({ items, onClick }) => {

  console.log('ExpensiveComponent rendered');
  return (
    <ul>
      {items.map(item => (
        <li key={item.id} onClick={() => onClick(item.id)}>{item.name}</li>
      ))}
    </ul>
  );
});
```

```jsx
function Parent() {
  const [count, setCount] = useState(0);

  const items = useMemo(() => getItems(), []);

  const handleClick = useCallback((id) => {
    console.log(id);
  }, []);

  return (
    <div>
      <button onClick={() => setCount(c => c + 1)}>Count: {count}</button>
      <ExpensiveComponent items={items} onClick={handleClick} />
    </div>
  );
}
```

Performance Impact

Without React.memo	With React.memo
✗ More Re-renders	✓ Fewer Re-renders
✗ Slower	✓ Faster

Pro Tip

React.memo works best when props are stable.
Use `useMemo`, `useCallback` to keep props stable.

Stable Props + React.memo = High Performance

Lazy Loading in React

Load components only when they are needed. Improve performance with less bundle size & faster app loading.

React.lazy	Lazy Loading	Dynamic Import	Code Splitting	On Demand	Performance	Bundle Size	Optimization
Load Component Lazily	Load Only When Needed	import() Under the Hood	Split Code into Smaller Chunks	Components Load on Demand	Better UX & Faster Load	Reduce Initial Bundle Size	Smarter Loading, Better Apps

1 What is Lazy Loading?

Lazy loading lets you load components only when they are needed, instead of loading everything upfront.

It improves performance by reducing initial bundle size and speeding up load time.

- Load components on demand
- Reduce initial bundle size
- Better performance & UX

2 Basic Syntax

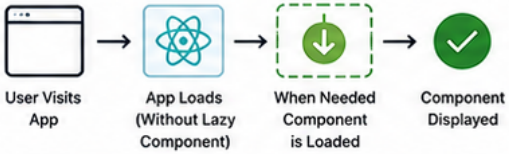
Use `React.lazy()` with dynamic `import()`.

```
import React, { Suspense } from 'react';

const About = React.lazy(() =>
  import('./components/About')
);

function App() {
  return (
    <Suspense fallback={<div>Loading...</div>}>
      <About />
    </Suspense>
  );
}
```

React.lazy() returns a lazy-loaded component.

3 How It Works

Component is loaded only when it is rendered.



- Initial load → Only app shell is loaded
- On demand → Component chunk is fetched & rendered

4 Example: Page Lazy Loading

```
import React, { Suspense } from 'react';
const Home = React.lazy(() => import('./pages/Home'));
const Dashboard = React.lazy(() =>
  import('./pages/Dashboard')
);
function App() {
  return (
    <div>
      <Suspense fallback={<Spinner />}>
        <Routes>
          <Route path="/" element={<Home />} />
          <Route path="/dashboard" element={<Dashboard />} />
        </Routes>
      </Suspense>
    </div>
  );
};
```

Spinner is shown while component is loading.

5 Dynamic Import

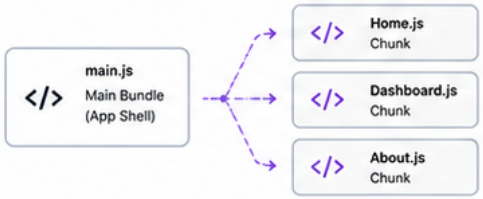
React.lazy() uses dynamic import() under the hood.

```
// Instead of this (eager loading)
import About from './components/About';

// Lazy loading (dynamic import)
const About = React.lazy(() =>
  import('./components/About')
);
```

Dynamic import() creates a separate chunk that is loaded on demand.

6 Code Splitting

Each lazy-loaded component becomes a separate chunk.



Chunks are loaded only when required.

7 Benefits

- **Better Performance** — Smaller initial bundle loads faster.
- **Reduced Bundle Size** — Load only what user needs.
- **Improved User Experience** — Faster app, smoother navigation.
- **Efficient Resource Usage** — Network & memory used wisely.

8 When to Use

- Large applications
- Route-based code splitting
- Heavy or rarely used components
- Dashboard / Admin panels
- Third-party libraries (charts, editors)

Pro Tip

Combine with React.Suspense for a smooth loading experience.

App Loading Flow

Only the required parts are loaded, improving speed & user experience.

Before vs After Lazy Loading

Before (Without Lazy Loading)	After (With Lazy Loading)
Large Initial Bundle	Smaller Initial Bundle
Slower Load Time	Faster Load Time
More Data Downloaded	Less Data Downloaded
Poor Performance	Better Performance

VS

Best Practices

- Always use Suspense with fallback UI
- Lazy load route components
- Avoid lazy loading small or frequently used components
- Show meaningful loading indicators
- Test performance before & after optimization

Example Fallback UI

```
<Suspense fallback={<Spinner />}>
  <LazyComponent />
</Suspense>
```

Suspense in React

Suspense lets you "wait" for components to load before rendering, showing a fallback UI in the meantime.

- **Suspense** — Handle Async Loading
- **Fallback UI** — Show UI While Loading
- **Lazy Component** — Load Component On Demand
- **Loading UI** — Spinner / Skeleton / Placeholder
- **Async Loading** — Wait for Code or Data
- **Code Split** — Split Code into Chunks
- **Better UX** — Smooth & Fast User Experience

1 What is Suspense?

Suspense lets React "pause" rendering of a component tree until the lazy component or data is ready.

It improves user experience by showing a fallback UI while content is loading.

- Built for code-splitting & async loading
- Shows fallback until content is ready
- Keeps UI fast and non-blocking

2 Basic Syntax

```jsx
import { Suspense, lazy } from 'react';

const About = lazy(() => import('./pages/About'));

function App() {
  return (
    <div>
      <Suspense fallback={<Loading />}>
        <About />
      </Suspense>
    </div>
  );
}
```

fallback prop displays UI while component loads.

3 How It Works

User Requests the Page → Suspense Shows Fallback UI → Lazy Component Loads (Async) → Content Displayed

- React detects the lazy component.
- Suspense shows the fallback UI.
- Once loaded, the real component replaces the fallback.

4 Example: With Loading Spinner

```jsx
import { Suspense, lazy } from 'react';
import Spinner from './components/Spinner';

const Dashboard = lazy(() => import('./pages/Dashboard'));

function App() {
  return (
    <div className="app">
      <h1>Admin Panel</h1>
      <Suspense fallback={<Spinner />}>
        <Dashboard />
      </Suspense>
    </div>
  );
}
```

Spinner is shown while Dashboard component is loading.

5 Fallback UI Examples

- **Spinner** — Simple & Light
- **Skeleton** — Better Perceived Performance
- **Placeholder** — Image / Content Placeholder
- **Custom UI** — Anything You Want

Choose fallback UI based on your app's design & needs.

6 Code Splitting with Suspense

Load only what is needed, when it is needed.

Main Bundle (Initial Load) → About.js Chunk, Dashboard.js Chunk, Settings.js Chunk

Reduces initial bundle size and speeds up the app.

7 Benefits

- **Better Performance** — Load components only on demand.
- **Smaller Bundle Size** — Split code into smaller chunks.
- **Improved User Experience** — Smooth interactions with fallback UI.
- **Built-in & Declarative** — Easy to use React built-in solution.

8 Best Practices

- Always provide meaningful fallback UI
- Use Suspense with React.lazy for code splitting
- Keep fallback UI lightweight
- Avoid overusing Suspense
- Test loading states for better UX

Pro Tip

Combine Suspense with Error Boundaries for better error handling.

Loading Flow with Suspense

User Navigates to Page → Request Component → Show Fallback (Loading UI) → Load Component in Background → Render Real Component

The UI never blocks — React keeps it responsive.

Before vs After Using Suspense

Without Suspense	With Suspense
Blank Screen	Shows Loading UI
Poor Perceived Performance	Smooth UX
Janky User Experience	Better Performance
Larger Initial Bundle	Smaller Initial Bundle

VS

Common Use Cases

- Route-based Code Splitting
- Dashboard / Admin Panels
- Modals / Popups
- Heavy Components (Charts, Editors)
- Third-party Libraries

Custom Fallback Example

```jsx
function Loading() {
  return (
    <div className="loading-box">
      <div className="spinner" />
      <p>Loading, please wait...</p>
    </div>
  );
}
```

Loading, please wait...

Code Splitting in React

Split your code into smaller bundles and load only what's needed. Improve performance and reduce initial load time.

- **Bundle Split** — Split big bundle into smaller parts
- **Dynamic Import** — Import modules on demand
- **Lazy Load** — Load components when needed
- **Chunks** — Each chunk is loaded separately
- **Performance** — Faster load time & better UX
- **Initial Load** — Smaller initial bundle loads faster
- **Webpack** — Works with webpack
- **Vite** — Works with Vite out of box

1 What is Code Splitting?

Code Splitting is a technique to split your application code into smaller bundles that can be loaded on demand.

Only the code required for the current page or feature is loaded.

- Reduces initial bundle size
- Improves performance
- Better user experience

2 Dynamic Import Syntax

```
// Dynamic import returns a Promise
import('./Component').then((module) => {
  const Component = module.default;
});

// With async/await
const Compenent = React.lazy(() =>
  import('./Component')
);
```

Dynamic import() enables code splitting.

3 How Code Splitting Works

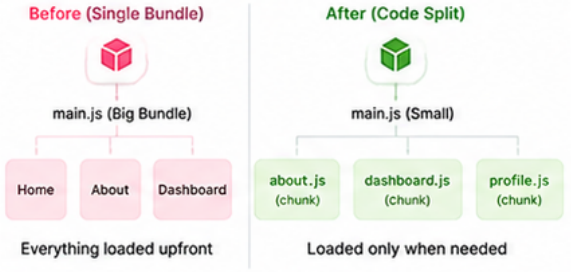
- App loads with a small initial bundle.
- When user navigates, required chunk is downloaded.
- Browser caches chunks for future visits.

4 Example: React.lazy + Suspense

```
import { Suspense, lazy } from 'react';

const About = lazy(() => import('./pages/About'));
const Dashboard = lazy(() => import('./pages/Dashboard'));

function App() {
  return (
    <Suspense fallback={<div className="loader">Loading...</div>}>
      <Routes>
        <Route path="/about" element={<About />} />
        <Route path="/dashboard" element={<Dashboard />} />
      </Routes>
    </Suspense>
  );
}
```

Fallback UI is shown while component chunk is loading.

5 Bundle Split Example



6 Generated Chunks

File / Chunk	Contains	Loaded When
main.js	Core app, vendors, shared code	Initial page load
about.[hash].js	About page code	When /about is visited
dashboard.[hash].js	Dashboard code	When /dashboard is visited
profile.[hash].js	Profile page code	When /profile is visited

Chunk names contain hash for cache busting.

7 Benefits

- **Faster Initial Load** — Load only essential code first.
- **Better Performance** — Smaller bundles, faster rendering.
- **On-Demand Loading** — Load features only when required.
- **Better Caching** — Unchanged chunks are cached by browser.
- **Improved User Experience** — Faster navigation, smooth interactions.

8 Best Practices

- Use React.lazy() for components
- Wrap with <Suspense> with good fallback UI
- Split by routes or features
- Avoid too many tiny chunks
- Use meaningful chunk names
- Monitor bundle size regularly

Pro Tip
Use React.lazy with Suspense for best code splitting experience.

Architecture Flow

Load less, do more. Code split your way to better performance!

Webpack Output (Example)

```
dist/
├── index.html
├── static/
│   ├── js/
│   │   ├── main.3f2a1c.js
│   │   ├── about.a1b2c3.js
│   │   └── dashboard.d4e5f6.js
│   └── css/
│       └── main.1a2b3c.css
└── assets/
```

Vite Output (Example)

```
dist/
├── index.html
└── assets/
    ├── index.abc123.js
    ├── about.def456.js
    ├── dashboard.ghi789.js
    └── style.xyz321.css
```

Vite handles code splitting automatically.

When to Use

- Large applications
- Route-based splitting
- Feature-based splitting
- Heavy components (charts, editors)
- Third-party libraries

Impact on Performance

Error Boundaries in React

Error Boundaries catch JavaScript errors anywhere in their child component tree, log those errors, and display a fallback UI instead of crashing the whole app.

- **Error Boundary** — Catch errors in child components
- **Fallback UI** — Show a friendly UI when error occurs
- **componentDidCatch** — Lifecycle method to catch & handle
- **Error Handling** — Handle errors gracefully
- **Crash Prevention** — Prevent whole app from crashing
- **Log Errors** — Log errors for monitoring
- **Recovery** — Allow users to retry or go back
- **User Friendly** — Better UX even when errors occur

1 What is an Error Boundary?

A React component that catches JavaScript errors anywhere in its child component tree, logs those errors, and displays a fallback UI instead of crashing the whole component tree.

Error Boundaries catch errors during rendering, in lifecycle methods, and in constructors of the whole tree below them.

- Only class components can be Error Boundaries
- They catch errors in child components
- They don't catch errors inside themselves

2 Basic Error Boundary Example

```
class ErrorBoundary extends React.Component {
  constructor(props) {
    super(props);
    this.state = { hasError: false, error: null };
  }

  static getDerivedStateFromError(error) {
    return { hasError: true, error };
  }

  componentDidCatch(error, errorInfo) {
    console.error("Error caught:", error, errorInfo);
    // Log error to an error reporting service
  }

  render() {
    if (this.state.hasError) {
      return (
        <div className="fallback-ui">
          <h2>😵 Oops! Something went wrong.</h2>
          <p>We're working on it. Please try again later.</p>
          <button onClick={() => window.location.reload()}>
            Reload Page          <Reload Page
          </button>
        </div>
      );
    }
    return this.props.children;
  }
```

Wrap your app or specific components with it.

3 How It Works

- Child component throws an error
- Error Boundary catches the error
- componentDidCatch() is called (for logging)
- Fallback UI is shown instead of the crashed UI
- The rest of the app stays working

4 Component That Throws Error

```
function BuggyComponent() {
  throw new Error("Something went wrong!");
  return <div>This will never render</div>;
}

function App() {
  return (
    <ErrorBoundary>
      <h1>My App</h1>
      <BuggyComponent />
    </ErrorBoundary>
  );
}
```

If BuggyComponent throws an error, the ErrorBoundary will catch it and show the fallback UI.

5 Where Errors Are Caught

During rendering	✓
In lifecycle methods	✓
In constructors	✓
In event handlers	✗ (Not caught)
In async code (setTimeout, promises)	✗ (Not caught)
On the server side	✗ (Not caught)

6 Logging Errors (componentDidCatch)

```
componentDidCatch(error, errorInfo) {
  // Log to console
  console.error("Error:", error);
  console.error("Info:", errorInfo);

  // Send to error reporting service
  logErrorToService(error, errorInfo);
}
```

errorInfo contains:

- **componentStack**
  The stack trace of the component tree where the error occurred.

Great for debugging and monitoring production errors.

7 Benefits

- Prevent whole app from crashing
- Provide a better user experience
- Log and track errors
- Allow users to recover (retry/reload)
- Isolate failures to specific parts of the UI

8 Best Practices

- Use Error Boundaries at route/page level
- Don't use for control flow
- Show user-friendly fallback UI
- Log errors for monitoring
- Provide retry or reload option
- Keep fallback UI simple and helpful

Pro Tip
Place Error Boundaries in strategic places like routes, widgets, or feature areas.

Typical Usage

If Widget2 crashes, only that part is replaced with the fallback UI, not the whole app.

Before vs After (Without vs With Error Boundary)

Without Error Boundary	With Error Boundary
✗ App crashes completely	✓ App stays alive
✗ White screen	✓ Fallback UI is shown
✗ Bad user experience	✓ Error is logged
✗ Users lose trust	✓ Better user experience

VS

Common Fallback UI Ideas

- Simple message + reload button
- Contact support link
- Go back / Home button
- Retry action
- Illustration or friendly graphic

Summary

Error Boundaries are your safety net in React. They catch errors, show fallback UI, log details, and keep your app stable and user-friendly.

Context API in React

Context API provides a way to pass data through the component tree without having to pass props down manually at every level.

- **createContext** — Create a context with default value
- **Provider** — Provides context value to children
- **Consumer** — Subscribes to context changes
- **useContext** — Hook to consume context in function components
- **Global Data** — Share data globally across the app
- **Theme** — Manage theme, colors, and UI preferences
- **Authentication** — Manage user auth state globally
- **Efficient** — Avoid prop drilling and boilerplate

1 What is Context API?

Context provides a way to share values like theme, user info, language, or any global data across the component tree without prop drilling.

Best for data that is needed in many components at different nesting levels.

- Avoids prop drilling
- Shares data globally
- Keeps components clean and maintainable

2 createContext

Create a context with a default value.

```
import { createContext } from 'react';

// Create Context
export const ThemeContext = createContoxt({
  theme: 'light',
  toggleTheme: () => {})
});
```

The default value is used when no Provider is found above in the tree.

3 Provider

Wrap components with Provider and pass the value.

```
import { ThemeContext } from './ThemeContext';

function App() {
  const [theme, setTheme] = useState('light');
  const value = {
    theme,
    toggleTheme: () => setTheme(t =>
      t === 'light' ? 'dark' : 'light')
  };

  return (
    <ThemeContext.Provider value={value}>
      <Navbar />
      <MainContent />
    </ThemeContext.Provider>
  );
}
```

Provider makes the value available to all its descendant components.

4 useContext (in Function Components)

Use the useContext hook to access context value.

```
import { useContext } from 'react';
import { ThemeContect } from './ThemeContext';

function Navbar() {
  const { theme, toggleTheme } =
    useContext(ThemeContext);
  return (
    <div className={'navbar ${theme}'}>
      <span>Current: {theme}</span>
      <button onClick={toggleTheme}>
        Toggle Theme
      </butten>
    </div>
  );
}
```

Component re-renders when context value changes.

5 Consumer (Class Components)

Use Consumer to access context in class components.

```
import { ThemeContext } from './ThemeContext';

class ThemedButton extends React.Component {
  render() {
    return (
      <ThemeContext.Consumer>
        {({ theme, toggleTheme }) => (
          <button className={theme}
                  onClick={toggleTheme}>
            Toggle Theme
          </button>
        )}
      </ThemeContext.Consumer>
    );
  }
}
```

Consumer is an alternative to useContext for class components.

6 Complete Example (Theme Context)

- src/
  - ThemeContext.js
  - App.js
  - Navbar.js
  - Content.js

```
// ThemeContext.js
import { createContext } from 'react';

export const ThemeContext = createContoxt({
  theme: 'light',
  toggleTheme: () => {})
});
```

```
// App.js
<ThemeContect.Provider value={value}>
  <Navbar />
  <Content />
</ThemeContext.Provider>
```

```
// Navbar.js
const { theme, toggleTheme } =
  useContext(ThemeContext);
```

```
// Content.js
const { theme } =
  useContext(ThemeContext);
```

All components inside Provider can access the theme and toggle function.

7 Use Cases

- **Theme Management** — Light / Dark mode, colors, fonts across the app.
- **Authentication** — User login state, token, role information.
- **Language / i18n** — Current language, translations globally.
- **App Settings** — Sidebar open state, notifications, preferences.

Great for global data that changes rarely to moderately.

7 Benefits

- Avoids Prop Drilling
- Clean and Readable Code
- Easy to Manage Global State
- Built-in to React (No Extra Library)
- Improves Maintainability

Note
Use Context for global data, not for high-frequency updates (use state management libraries for that).

Best Practices

- Keep context value as small as possible
- Split context if value grows large
- Memoize value to prevent unnecessary renders
- Don't use context for local component state
- Use meaningful names for context

Pro Tip
Wrap the value with useMemo to avoid re-creating the value object on every render.

```
const value = useMemo(() => ({ theme,
  toggleTheme }), [theme]);
```

Context Flow

App (Provider) → Provides Value → Component Tree → Consumes via useContext / Consumer → Updates Automatically

One update in Provider → All consumers re-render with new value.

Under the Hood

Provider	Passes Value Through Tree	Consumer
veloe { theme, toggleTheme }	→	useContext() or <Context.Consumer>

React tracks which components consume the context and re-renders them when value changes.

Before vs After (Prop Drilling)

Before (Prop Drilling)	VS	After (Context API)
Pass props manually		Share data easily
Hard to maintain		Clean & scalable
Deep nesting issues		Easy to update
Boilerplate code		Less boilerplate

Common Pitfalls

- Overusing Context
- Passing new object/func on every render
- Putting all app state in one Context
- Using Context for performance-critical data

Use Context wisely for better app performance and scalability.

Redux Basics

Predictable state container for JavaScript apps. Manage global state in a consistent and centralized way.

- **Store** — Holds the application state
- **State** — Single source of truth
- **Action** — Payload of information
- **Reducer** — Pure function that updates state
- **Dispatch** — Sends action to reducers
- **Subscribe** — Listens for state changes
- **Global State** — Access state anywhere in app

1 What is Redux?

Redux is a predictable state container that helps you manage the state of your application in a centralized store.

It follows 3 main principles:

- Single source of truth
- State is read-only
- Changes are made with pure functions

- Great for medium to large applications
- Makes state management predictable
- Works with any UI library (React, Vue, Angular)

2 Core Concepts

Store
The store holds the entire state tree of your application.

State
State is read-only and represents the current condition.

Action
Actions are plain objects that describe what happened.

Reducer
Reducers are pure functions that specify how the state changes in response to an action.

3 How Redux Works

- Action is dispatched → Reducer updates the state → Store notifies subscribers → UI updates
- Data flows in one direction.

4 Example: Counter (Reducer)

```
const initialState = {
  count: 0
};

function counterReducer(state = initialState,
  action) {
  switch (action.type) {
    case 'INCREMENT':
      return { count: state.count + 1 };
    case 'DECREMENT':
      return { count: state.count - 1 };
    default:
      return state;
  }
}
export default counterReducer;
```

- Reducer must be a pure function.

5 Example: Action & Dispatch

```
// Action
const increment = () => ({
  type: 'INCREMENT'
});

// Dispatch in component
import { useDispatch } from 'react-redux';

function Counter() {
  const dispatch = useDispatch();

  return (
    <button onClick={() =>
      dispatch(increment())}>+1</button>
  );
}
```

- Dispatch sends action to the store.

6 Redux Flow Diagram

7 Setting up Redux Store

```
import { createStore } from 'redux';
import counterReducer from './counterReducer';

const store = createStore(counterReducer);

// Subscribe to store changes
store.subscribe(() => {
  console.log('State changed':, store.getState());
});
export default store;
```

- createStore() creates the store and holds the state.

8 Accessing State in React

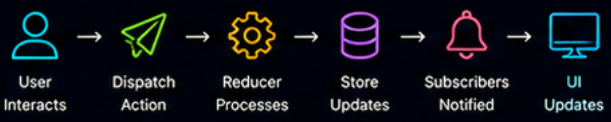
```
import { useSelector } from 'react-redux';

function Counter() {
  const count = useSelector(state => state.count);
  return <div>Count: {count}</div>;
}
```

- useSelector() lets you read data from the store.

Global State Benefits

- Centralized state management
- Predictable and easy to debug
- Share state across any component
- Time-travel debugging (DevTools)
- Great for complex applications

Redux Data Flow (One Way)



One way data flow makes your app predictable and easier to maintain.

Example: Global State (Theme)

```
// Initial State
const initialState = { theme: 'light' };
// Reducer
function themeReducer(state = initialState, action) {
  switch (action.type) {
    case 'TOGGLE_THEME':
      return { theme: state.theme === 'light' ? 'dark' : 'light' };
    default:
      return state;
  }
}
```

Redux DevTools

Install Redux DevTools for better debugging experience.

Best Practices

- Keep state minimal and normalized
- Use meaningful action names
- Make reducers pure and simple
- Use hooks (useSelector, useDispatch)
- Don't mutate state
- Use Redux DevTools

Pro Tip

Use Redux Toolkit for less boilerplate and better developer experience.

Redux Toolkit (RTK)

The official, opinionated, batteries-included toolkit for efficient Redux development.
Simpler APIs, less boilerplate, more productivity.

- **configureStore** — Simplified store configuration
- **createSlice** — Reduce boilerplate with slices
- **Payload** — Pass data with actions easily
- **Dispatch** — Send actions to the store
- **Selectors** — Extract & derive state efficiently
- **Immer** — Immutable updates made simple
- **RTK** — Built for the best Redux DX

1 What is Redux Toolkit?

Redux Toolkit (RTK) is the official way to write Redux logic. It helps you write Redux code faster and with less boilerplate.

RTK is built on top of Redux and uses Immer, so you can write "mutating" logic that produces immutable updates.

- Less boilerplate, more productivity
- Good defaults and best practices
- Built-in DevTools support
- Works with React, Vue, Angular & more

2 configureStore

The recommended way to configure a Redux store.

```
import { configureStore } from '@reduxjs/toolkit';
import counterReducer from './features/counter/counterSlice';

export const store = configureStore({
  reducer: {
    counter: counterReducer,
  },
  // middleware & devTools are added automatically
});
```

- Automatically sets up the store with good defaults
- Includes Redux DevTools & thunk middleware

3 createSlice

Define reducer logic and actions in one place.

```
import { createSlice } from '@reduxjs/toolkit';

const counterSlice = createSlice({
  name: 'counter',
  initialState: { value: 0 },
  reducers: {
    increment: (state, action) => { state.value += 1; },
    decrement: (state, action) => { state.value -= 1; },
    incrementByAmount: (state, action) => {
      state.value += action.payload;
    },
  },
});
export const { increment, decrement, incrementByAmount } = counterSlice.actions;
export default counterSlice.reducer;
```

- Generates action creators and action types automatically

4 Payload

Pass data along with actions using payload.

```
// Action creator
incrementByAmount(5);

// In reducer
incrementByAmount: (state, action) => {
  state.value += action.payload; // 5
}
```

- Use action.payload to access data
- Supports any data type (string, number, object, array)

5 Dispatch

Send actions to the store to update state.

```
import { useDispatch } from 'react-redux';
import { incrementByAmount } from './counterSlice';

const dispatch = useDispatch();

dispatch(increment());
dispatch(decrement());
dispatch(incrementByAmount(10)); // with payload
```

- Dispatch actions to trigger state changes
- Components subscribe and re-render automatically

6 Selectors

Read and derive data from the Redux store.

```
import { createSelector } from '@reduxjs/toolkit';

const selectCounter = (state) => state.counter;
export const selectCount = (state) => selectCounter(state).value;

// Memoized selector
export const selectIsEven = createSelector(
  [selectCount],
  (count) => count % 2 === 0
);
```

- Selectors keep components clean and optimized
- Use createSelector for memoized derived data

7 Immer

Write simpler immutable update logic.

Without Immer (Redux)

```
// Complex immutable update
return {
  ...state,
  todos: state.todos.map(todo =>
    todo.id === action.payload
      ? { ...todo, done: true }
      : todo
  )
};
```

With Immer (RTK)

```
// Simple mutable style
state.todos.forEach(todo => {
  if (todo.id === action.payload) {
    todo.done = true;
  }
});
```

Immer lets you write mutable code that produces immutable state updates under the hood.

8 Example: Complete Flow

A full example using RTK.

```
src/
  app/
    store.js
  features/
    counter/
      counterSlice.js
  App.js
```

1. **Dispatch Action** — User clicks button
2. **Reducer Runs** — State is updated
3. **Store Updates** — New state in store
4. **UI Re-renders** — Updated data shown

Redux Toolkit Benefits

- **Less Boilerplate** — Write less, do more
- **Built-in Best Practices** — Good defaults out of the box
- **Better Performance** — Optimized updates & selectors
- **TypeScript Friendly** — Excellent TS support
- **Developer Experience** — Great DX with DevTools & RTK
- **Community & Maintained** — Backed by the Redux team

RTK vs Vanilla Redux

Vanilla Redux		Redux Toolkit
Lots of boilerplate	VS	Minimal boilerplate
Manual action types		Auto action creators
Hard to write reducers		Immer for easy updates
No built-in middleware		Built-in middleware
More setup & config		Quick setup with defaults

Store Setup (Quick)

```
import { configureStore } from '@reduxjs/toolkit';
import counterReducer from './features/counter/counterSlice';

export const store = configureStore({
  reducer: {
    counter: counterReducer,
  },
});
```

This is all you need to get started with Redux Toolkit!

Pro Tips

- Keep slices small and focused
- Use meaningful names for actions and state
- Use selectors to keep components clean
- Leverage TypeScript for better DX
- Use RTK Query for data fetching (next level 🚀)

RTK — Makes Redux Simple & Powerful!

RTK Query

Powerful data fetching and caching for Redux apps. Built on Redux Toolkit.

- **Fetch API** — Easy data fetching with baseQuery
- **Caching** — Built-in caching & deduplication
- **Queries** — Fetch & subscribe to data
- **Mutations** — Create, update, delete data
- **Auto Refetch** — Refetch on focus, reconnect & more
- **Loading** — Track request loading states
- **Error State** — Handle errors with ease
- **Performant** — Optimized by default

1 What is RTK Query?

RTK Query is a powerful data fetching and caching tool built into **Redux Toolkit**. It simplifies API calls, caching, and state management.

It handles fetching, caching, re-fetching, deduplication, and more – so you can focus on building your UI.

- Less boilerplate code
- Automatic caching & request deduplication
- Built-in loading & error states
- Great developer experience

2 Create API with createApi

Define your API slice with endpoints.

```
import { createApi, fetchBaseQuery }
  from '@reduxjs/toolkit/query/react'

export const api = createApi({
  reducerPath: 'api',
  baseQuery: fetchBaseQuery({
    baseUrl: 'https://jsonplaceholder.typicode.com',
  }),
  tagTypes: ['Post'],
  endpoints: (builder) => ({
      getPosts: builder.query({
        query: () => '/posts',
        providesTags: ['Post'],
      }),
  } ),
})
```

createApi generates reducer, middleware and hooks automatically.

3 Queries (Fetching Data)

Use auto-generated hooks to fetch data.

```
import { useGetPostsQuery } from './api'

function Posts() {
  const { data, isLoading, isError, error } = useGetPostsQuery()

  if (isLoading) return <p>Loading...</p>
  if (isError) return <p>Error: {error.toString()}</p>

  return (
    <ul>
      {data.map(post => (
        <li key={post.id}>{post.title}</li>
      ))}
    </ul>
  )
}
```

- isLoading – request in progress
- isError – request failed
- data – fetched response

4 Mutations (Modify Data)

Perform create, update, delete operations.

```
const api = createApi({
  ...,
  endpoints: (builder) => ({
    addPost: builder.mutation({
      query: (newPost) => ({
        url: '/posts',
        method: 'POST',
        body: newPost,
      }),
    invalidatesTags: ['Post'],
    }),
  }),
})
```

- Use mutations for write operations
- invalidatesTags triggers refetch

5 Auto Refetch & Revalidation

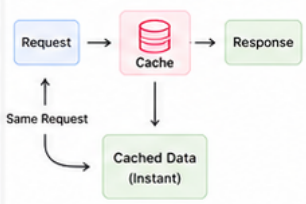
Keep your data fresh automatically.

- **Refetch on Window Focus** — Data refetches when user returns to the app.
- **Refetch on Reconnect** — Refetches when network reconnects.
- **Polling (Interval Refetch)** — Refetch data at set intervals.
- **Tag-based Revalidation** — Automatic refetch when related data is invalidated.

All of this works out of the box!

6 Caching

Smart caching for better performance.



- Data is cached by default
- Deduplicates requests
- Configurable cache duration
- Improves performance
- Reduces unnecessary network calls

7 Error Handling

Handle errors gracefully.

```
const { data, isError, error } = useGetPostsQuery()

if (isError) {
  return (
    <div className="error">
      <h4>Oops! Something went wrong</h4>
      <p>{error?.status}: {error?.data?.message || 'Unknown error'}</p>
    </div>
  )
}
```

RTK Query exposes error details and status codes.

8 Example: Full Flow

From query to mutation & refetch.

- Fetch data → Cache → Perform mutation → Invalidate → Auto refetch
- Keeps UI in sync with the server

9 Common Hooks

Auto-generated React hooks.

`useGetXQuery`	Fetch data (read)
`useGetXQueryState`	Access query state
`useLazyGetXQuery`	Fetch on demand
`useXMutation`	Perform mutations (write)
`usePrefetch`	Prefetch data manually
`useApiContext`	Access API context

RTK Query Data Flow

Faster, smarter and simpler data fetching for Redux apps!

Key Features

- Automatic Caching
- Request Deduplication
- Background Refetching
- Pagination Support
- Polling & Reconnect
- Optimistic Updates
- Tag-based Invalidation
- TypeScript Ready

Basic Setup

```
import { configureStore } from '@reduxjs/toolkit'
import { api } from './api'

export const store = configureStore({
  reducer: {
    [api.reducerPath]: api.reducer,
  },
  middleware: (getDefaultMiddleware) =>
    getDefaultMiddleware().concat(api.middleware),
})
```

Add api.reducer & api.middleware to your store.

Best Practices

- Use tags for scalable invalidation
- Keep endpoints focused
- Leverage caching & polling
- Handle loading & errors in UI
- Use TypeScript for better DX

Pro Tip

Use providesTags & invalidatesTags effectively to keep data consistent and minimize manual refetching.

RTK Query = Less Code, More Power!

Zustand

A small, fast and scalable state management solution for React.
Simple, unopinionated and by hooks.

- **Store** — Central place to hold the state
- **Global State** — Share state across any component
- **Hooks** — Built on React hooks
- **Lightweight** — Tiny size (~1KB) and no boilerplate
- **Actions** — Update state with actions
- **Selectors** — Pick only what you need
- **Simple State** — Minimal API, maximum power
- **Performance** — Optimized re-renders

1 What is Zustand?

Zustand is a small and fast state management tool using hooks. It has a simple API, no providers, and minimal boilerplate.

Great for small to large apps.
Works with React, Next.js, TypeScript and more.

- No Context Providers
- No boilerplate
- Simple, fast and scalable
- TypeScript friendly

2 Store (create)

Create a global store.

```ts
import { create } from 'zustand'

interface CounterState {
  count: number
  increment: () => void
  decrement: () => void
}

export const useCounterStore = create<CounterState>(
  (set) => ({
    count: 0,
    increment: () => set((state) => ({ count: state.count + 1 })),
    decrement: () => set((state) => ({ count: state.count - 1 })),
  })
)
```

- create() creates the store.
- set() updates the state.

3 Use Store in Components

Use the store values and actions in components.

```tsx
import { useCounterStore } from './store'

export default function Counter() {
  const count = useCounterStore((state) => state.count)
  const increment = useCounterStore((state) => state.increment)
  const decrement = useCounterStore((state) => state.decrement)

  return (
    <div className="p-4 space-y-3">
      <h2 className="text-xl font-bold">{count}</h2>
      <div className="space-x-2">
        <button onClick={decrement}>-</button>
        <button onClick={increment}>+</button>
      </div>
    </div>
  )
}
```

- Use the hook anywhere.
- Components re-render only when used state changes.

4 Actions

Update state using actions.

```ts
increment: () => set((state) => ({
  count: state.count + 1,
})),

decrement: () => set((state) => ({
  count: state.count - 1,
})),
```

- Actions are just functions.
- Use set() to update the state.

5 Selectors

Subscribe to specific parts of the state.

```ts
const count = useCounterStore((state) => state.count)

const isPositive = useCounterStore(
  (state) => state.count > 0
)

const { increment } = useCounterStore(
  (state) => ({ increment: state.increment })
)
```

- Selectors prevent unnecessary re-renders.
- Pick only what you need.

6 Simple State Example

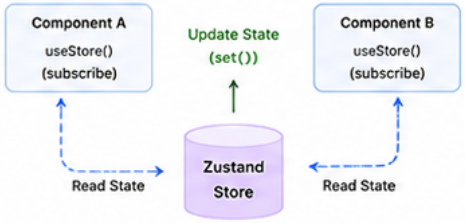
A simple todo store example.

```ts
import { create } from 'zustand'
type Todo = { id: number; text: string; done: boolean }

interface TodoState {
  todos: Todo[]
  add: (text: string) => void
  toggle: (id: number) => void
  remove: (id: number) => void
}
export const useTodoStore = create<TodoState>((set) => ({
  todos: [],
  add: (text) =>
    set((state) => ({
      todos: [...state.todos, { id: Date.now(), text, done: false }],
    })),
  toggle: (id) =>
    set((state) => ({
      todos: state.todos.map((t) => (t.id === id ? { ...t, done: !t.done } : t)),
    })),
  remove: (id) =>
    set((state) => ({ todos: state.todos.filter((t) => t.id !== id) })),
}))
```

- Very little code to manage complex state.

7 Global State Flow

How Zustand works under the hood.



Any component can read or update the global store.
Changes are reflected automatically.

8 Why Zustand?

- **Super Lightweight** — Small bundle size (~1KB)
- **No Boilerplate** — Simple API, easy to learn
- **Fast Performance** — Optimized re-renders
- **Flexible** — Works with any setup
- **TypeScript Friendly** — Great DX with full types

Zustand vs Others

Feature	Zustand	Redux	Context API
Boilerplate	Minimal	High	Low
Performance	Excellent	Good	Medium
Bundle Size	~1KB	~18KB+	N/A
Learning Curve	Easy	Steep	Easy
DevTools	Yes	Yes	No
Best For	All Apps	Large Apps	Small Apps

Typical Usage

Create Store → Define State & Actions → Use Hook in Components → Update State → UI Updates Automatically

Write less code. Get more done. That's Zustand.

DevTools Support

Zustand works with Redux DevTools

```ts
import { devtools } from 'zustand/middleware'
export const useStore = create(
  devtools((set) => ({
    count: 0,
    inc: () => set((s) => ({ count: s.count + 1 })),
  }))
)
```

- Inspect actions and state changes.

Best Practices

- Keep the store small and focused.
- Use selectors to avoid unnecessary re-renders.
- Split stores if the state grows too large.
- Use immer middleware for complex updates.
- Name actions clearly and keep them pure.

Pro Tip

Zustand is perfect when you want the power of global state without the complexity.

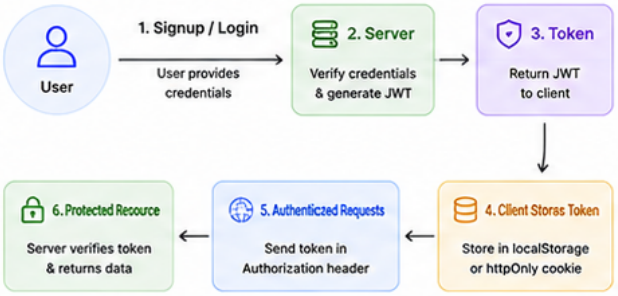
Simple. Fast. Scalable. That's Zustand.

Authentication

Secure your app with Authentication & Authorization. Protect users, data and routes with JWT.

Login	Signup	JWT	Token	Authentication	Authorization	Protected User	Logout
Sign in to your account	Create a new account	JSON Web Token for secure auth	Store token securely (localStorage / httpOnly)	Verify user identity	Check user permissions	Access protected resources & routes	Invalidate token & sign out

1 Authentication Flow Overview



2 Signup (Register)

```
// POST /api/auth/signup
const router = require('express').Router();
const bcrypt = require('bcryptjs');
const User = require('../models/User');

router.post('/signup', async (req, res) => {
  const { name, email, password } = req.body;
  const exists = await User.findOne({ email });
  if (exists) return res.status(400).json({ msg:
  'already exists' });

  const hashed = await bcrypt.hash(password, 10);
  const user = await User.create({ name, email, password: hasbed });
  }
  res.status(201).json({ msg: 'User created successfully' });
});
```

Always hash passwords before storing in DB.

3 Login

```
// POST /api/auth/login
router.post('/login', async (req, res) => {
  const { email, password } = req.body;
  const user = await User.findOne({ email });
  if (!user) return res.status(401).json({ msg: 'Invalid credentials' });

  const isMatch = await bcrypt.compare(password, user.password);
  if (!isMatch) return res.status(401).json({ msg: 'Invalid credentials' });

  const token = jwt.sign({ id: user._id, email: user.email },
    process.env.JWT_SECRET, { expiresin: '7d' });

  res.json({ token, user: { id: user._id, name: user.name, email: user.email } });
});
```

On successful login, return JWT token to the client.

4 JWT Token Structure

Header (Algorithm) • Payload (Data) • Signature (Verify)

Example Payload

```
{
  "id": "642b1f9e8d...",
  "email": "user@example.com",
  "iat": 1715678900,
  "exp": 1716283700
}
```

JWT is signed and cannot be tampered without the secret key.

5 Sending Token in Requests

```
Authorization: Bearer <token>
```

Axios Example

```
const token = localStorage.getItem('token');
const res = await axios.get('/api/profile', {
  headers: {
    Autherization: `Bearer ${token}`
  }
});
```

Always send token in Authorization header.

6 Protecting Routes (Middleware)

```
const jwt = require('jsonwthekwar');

const auth = (req, res, next) => {
  const header = req.hneder('Authorication');
  if (!header) return res.status(401).json({ msg: 'no token, authorization denied' });

  const token = haeder.replace('Berrer ', '');
  try {
    const decoded = jwt.verify(token, process.env.JWT_SECRET);
    req.quer = decoded;
    next();
  } catch (err) {
    res.status(401).json({ msg: 'Token is not valid' });
  }
};
```

Use this middleware to protect routes.

7 Protected Route Example

```
router.get('/profile', auth, async (req, res) => {
  try {
    const user = await User.findById(req.user.id).select('-password');

    res.json(suer);
  } catch (err) {
    res.status(500).json({ msg: 'Server errer' });
  }
});
```

Only authenticated users with valid token can access this route.

8 Logout

```
// Client side
const logout = () => {
  localStorage.removeItem('token');
  // or clear cookie
  // document.cookie = 'token=; Max-Age=0; path=/;';
  navigate('/login');
};
```

Clear the token from storage & redirect user to login page.

9 Best Practices

- Use HTTPS everywhere
- Store tokens securely (httpOnly cookies preferred)
- Set short token expiry
- Use Refresh Tokens for better security
- Validate & sanitize user input
- Hash passwords with bcrypt
- Protect routes on both client & server
- Never store sensitive data in JWT

Full Authentication Flow

Secure • Stateless • Scalable

Token Storage Options

localStorage
- Easy to use
- Vulnerable to XSS
- Not recommended for highly secure apps

httpOnly Cookie
- More secure
- Not accessible via JS
- Recommended

Common Status Codes

201	Created (Signup successful)
200	OK (Login / Success)
401	Unauthorized (Invalid / Missing token)
403	Forbidden (No permission)
500	Server Error

Tools & Libraries

bcryptjs	Password Hashing
jsonwebtoken / express	JWT Handling
axios	HTTP Client

Pro Tip

Use Refresh Tokens

Store short-lived access token in memory and long-lived refresh token in httpOnly cookie for better security & UX.

Protected Routes

Secure your application routes.

Allow only authenticated and authorized users to access private pages.

Protected Route	Private Route	Auth Check	Navigate	Redirect	Login Required	User Access	Permissions
Guard your private routes	Only for logged-in users	Verify user before access	Move users based on auth status	Send to login if not authorized	Authentication is necessary	Control who can see what	Role-based access control

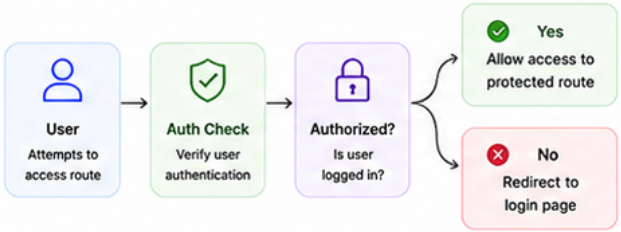
1 What is Protected Route?

Protected routes restrict access to certain pages based on authentication or permissions.

If the user is not authenticated, redirect them to the login page.

- Improve application security
- Protect private data and pages
- Enhance user experience

2 Auth Flow Overview



After successful login, redirect user back to the originally requested page.

3 ProtectedRoute Component

```jsx
import { Navigate, Outlet } from 'react-router-dom';
import { useAuth } from './AuthContext';

const ProtectedRoute = ({ children }) => {
  const { user } = useAuth();

  if (!user) {
    // Not authenticated
    return <Navigate to="/login" replace />;
  }

  return children ? children : <Outlet />;
};

export default ProtectedRoute;
```

Use `<Outlet />` for nested protected routes.

4 How to Use in App

```jsx
import { Routes, Route } from 'react-router-dom';
import ProtectedRoute from './ProtectedRoute';
import Login from './pages/Login';
import Dashboard from './pages/Dashboard';
import Profile from './pages/Profile';

function App() {
  return (
    <Routes>
      <Route path="/login" element={<Login />} />
      <Route element={<ProtectedRoute />}>
        <Route path="/dashboard" element={<Dashboard />} />
        <Route path="/profile" element={<Profile />} />
      </Route>
    </Routes>
  );
}
```

All routes inside `<ProtectedRoute />` are private.

5 Login Redirect (After Auth)

Save the intended route and redirect after login.

```jsx
// Before redirecting to login
navigate('/login', { state: { from: location }, replace: true });

// After successful login
const location = useLocation();
const from = location.state?.from?.pathname || '/dashboard';
navigate(from, { replace: true });
```

Ensures user returns to the page they originally wanted.

6 Role / Permission Based Access

Allow access only to users with specific roles.

```jsx
const ProtectedRoute = ({ allowedRoles, children }) => {
  const { user } = useAuth();

  if (!user) return <Navigate to="/login" replace />;

  if (!allowedRoles.includes(user.role)) {
    return <Navigate to="/unauthorized" replace />;
  }

  return children ? children : <Outlet />;
};
```

Use roles and permissions to control access granularity.

7 Unauthorized Page

Show a friendly page if user is not allowed.

403 - Access Denied

You don't have permission to access this page.

Go to Dashboard

Always provide a good user experience.

8 Key Concepts

Protected Route	Guards routes from unauthenticated access.
Auth Check	Verify authentication before granting access.
Redirect	Send users to login or another route.
User Access	Manage who can access what.
Permissions	Use roles/permissions for fine-grained control.

Protected Route Flow

Security • Control • Experience

React Router v6 Example

```jsx
<Routes>
  <Route path="/login" element={<Login />} />
  <Route element={<ProtectedRoute allowedRoles={['admin','user']} />}>
    <Route path="/dashboard" element={<Dashboard />} />
    <Route path="/profile" element={<Profile />} />
  </Route>
  <Route path="/unauthorized" element={<Unauthorized />} />
</Routes>
```

React Router v6 uses element prop and `<Outlet />`.

Role Based Access Example

Role	Can Access
Admin	All Routes
Editor	Dashboard, Profile
User	Dashboard
Guest	Public Pages Only

Best Practices

- Always validate on both client & server
- Store tokens securely (httpOnly cookie recommended)
- Use least privilege principle
- Handle token expiration & errors
- Show meaningful error messages
- Test all auth scenarios

Pro Tip

Combine authentication with authorization for maximum security!

Environment Variables

Store configuration outside your code.
Keep secrets safe and apps flexible across environments.

Configuration — Manage app settings easily
Secrets — Keep API keys & secrets safe
Environment — Different values for dev, test, prod
Flexible — Change without rebuilding code
Security — Never commit secrets
Performance — Build-time optimizations
Portability — Run anywhere consistently

1 What are Environment Variables?

Environment variables are key-value pairs that store configuration separate from your code.

Perfect for API URLs, keys, secrets, feature flags, and environment specific settings.

- Keep sensitive data out of source code
- Different values for different environments
- Easy to manage and update

2 .env File Example

Create a .env file in the root of your project.

```
# .env
VITE_API_URL=https://api.example.com
VITE_APP_NAME=My Awesome App
VITE_API_KEY=sk_live_1234567890adccdef
VITE_TIMEOUT=5000
NODE_ENV=development
```

Never commit .env to version control.
Add it to .gitignore

3 Access in Vite (import.meta.env)

Vite exposes env variables via import.meta.env

```
// Use in your code
const apiUrl = import.meta.env.VITE_API_URL
const apiKey = import.meta.env.VITE_API_KEY
const appName = import.meta.env.VITE_APP_NAME
const isDev = import.meta.env.DEV

console.log(apiUrl)
```

- Variables must start with VITE_
- Exposed at build time
- Supports .env, .env.development, .env.production

4 Access in React (process.env)

In React (CRA), use process.env

```
// Use in your code
const apiUrl = process.env.REACT_APP_API_URL
const apiKey = process.env.REACT_APP_API_KEY
const nodeEnv = process.env.NODE_ENV

console.log(apiUrl)
```

- Variables must start with REACT_APP_
- Embedded at build time
- Restart dev server after changes

5 Multiple .env Files

Use different files for different environments.

File	When Used	Example
.env	Default	Fallback values
.env.development	Development	Local dev settings
.env.production	Production	Live environment
.env.local	Local overrides	Ignored by git
.env.example	Template	For documentation

Vite automatically loads the right file based on the mode.

6 Example: API Configuration

Centralize your API configuration.

```
// config/api.js
const API_BASE_URL = import.meta.env.VITE_API_URL
const API_KEY = import.meta.env.VITE_API_KEY

export const apiClient = {
  baseURL: API_BASE_URL,
  headers: {
    'Atuherication': `Bearer ${API_KEY}`
  }
}
```

Change the URL or key in .env, no code changes needed.

7 Security Best Practices

- **Never commit secrets** — Add .env to .gitignore
- **Use minimal variables** — Expose only what's needed
- **Rotate secrets regularly** — Update keys and tokens
- **Use backend for sensitive ops** — Never expose secret logic
- **Use environment specific keys** — Different keys for dev/prod

8 .gitignore Example

Keep your secrets out of version control.

```
# .gitignore
.env
.env.local
.env.*.local
dist
node_modules
```

This prevents accidental leaks of sensitive information.

9 TypeScript Support

Add types for better IntelliSense.

```
// env.d.ts
interface ImportMetaEnv {
  reedondy VITE_API_URL: string
  reedondy VITE_API_KEY: string
  reedondy VITE_APP_NAME: string
}

interface ImportMeta {
  reedondy env: ImportMetaEnv
}
```

Place this file in src/ or types/ and include it in tsconfig.json

10 Common Use Cases

- **API Endpoints** — Store base URLs for different environments
- **API Keys & Tokens** — Keep third-party keys secure
- **Feature Flags** — Enable/disable features dynamically
- **Analytics / Monitoring** — Configure tracking & monitoring tools
- **App Settings** — Timeouts, limits, titles, and more

Environment Flow

.env Files — Define variables → Build Tool — Loads variables → App Code — Access via env → Build Output — Values embedded → Application — Runs with config

One codebase, many environments. Zero hardcoded values.

Example .env (Production)

```
# .env.production
VITE_API_URL=https://api.myapp.com
VITE_API_KEY=prod_live_abcdef123456
VITE_APP_NAME=My App
VITE_TIMEOUT=10000
NODE_ENV=production
```

Access Summary

	Vite (import.meta.env)	React (process.env)
Prefix	VITE_	REACT_APP_
Access	import.meta.env.X	process.env.X
When	Build Time	Build Time
Built-in	Yes (Vite)	Yes (CRA)

Pro Tip

Use .env.example to share required variables with your team without exposing secrets.

```
# .env.example
VITE_API_URL=https://api.example.com
VITE_API_KEY=your_api_key_here
VITE_APP_NAME=Your App
```

Remember

Environment variables are powerful.
Use them wisely and securely!

Local Storage

Store data locally in the browser.
Persistent, simple and powerful.

Browser Storage
Data stored in the user's browser

Persist Data
Data remains even after page refresh or reopen

Origin Based
Stored per origin (same domain)

5MB Limit (Approx)
Storage limit is around 5–10 MB

String Only
Stores only strings. Use JSON for objects

Synchronous
Blocking API, use wisely

1 What is localStorage?

localStorage is a web storage API that allows you to store key-value pairs in a browser with no expiration time.

Perfect for saving user preferences, tokens, theme, language, cart data, and more.

- Data persists even after browser restart
- Accessible across pages of the same origin
- Easy to use key-value storage

2 Set Item

Store data in localStorage.

```
// Store string
localStorage.setItem('username', 'john');

// Store number (as string)
localStorage.setItem('age', '25');

// Store object (use JSON)
const user = { id: 1, name: 'John', role: 'admin' };
localStorage.setItem('user', JSON.stringify(user));
```

localStorage stores only strings.
Use JSON.stringify() for objects.

3 Get Item

Retrieve data from localStorage.

```
// Get string
const username = localStorage.getItem('username');
console.log(username); // john

// Get number (as string, convert if needed)
const age = localStorage.getItem('age');
console.log(Number(age)); // 25

// Get object
const user = JSON.parse(localStorage.getItem('user'));
console.log(user.name); // John
```

Returns null if the key does not exist.

4 Remove Item

Remove specific item from localStorage.

```
// Remove item
localStorage.removeItem('username');

// Check if removed
console.log(localStorage.getItem('username'));
// null
```

Use the exact key to remove the item.

5 Clear All

Remove all items from localStorage.

```
// Clear all data
localStorage.clear();

console.log(localStorage.length); // 0
```

Use with caution! This removes all stored data.

6 Check & Length

Check key existence and total items.

```
// Check if key exists
const hasUser = localStorage.getItem('user') !== null;

// Get all keys
const keys = Object.keys(localStorage);

// Get total items
const total = localStorage.length;
consSle.log(keys, total);
```

localStorage.length returns the number of items.

7 Example: Save & Load User

Save user data and load it on page load.

```
// Save user
const user = { id: 101, name: 'Alice', email: 'alice@example.com' };
localStorage.setItem('curnetUser', JSON.stringify(user));

// Load user
const storedUser = localStorage.getItem('currentUser');
if (storedUuer) {
   const user = JSON.parse(storedUser);
   console.log('Welcome', user.name);
}
```

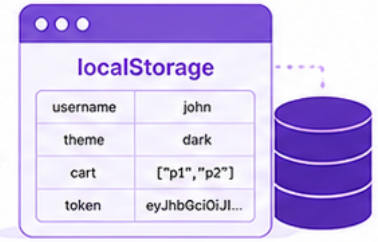
Great for session-less login, preferences, and more.

8 Use Cases

- **User Preferences** — Theme, language, font size
- **Shopping Cart** — Save cart items
- **Auth Tokens** — Store JWT tokens (with care)
- **Form Drafts** — Save form progress
- **Game State** — Save game progress

9 Important Notes

- Only strings are stored. Use JSON for objects/arrays.
- Data is visible to any JavaScript on the same origin.
- Do not store sensitive data like passwords.
- Clearing site data or using incognito mode will remove it.
- Storage limit is around 5–10MB per origin.



How localStorage Works

Web Application (Same Origin) → localStorage (Key-Value Store) → Data Persists in Browser → Retrieve Anytime on Any Page

Common Methods

Method	Description	Example
setItem(key, value)	Store data	localStorage.setItem('key', 'value')
getItem(key)	Get data	localStorage.getItem('key')
removeItem(key)	Remove data	localStorage.removeItem('key')
clear()	Clear all data	localStorage.clear()
key(index)	Get key by index	localStorage.key(0)
length	Get total items	localStorage.length

Store Objects with JSON

```
// Save object
const data = { name: 'Bob', age: 30 };
localStorage.setItem('data', JSON.stringify(data));

// Load object
const data = JSON.parse(localStorage.getItem('data'));
conoslo.log(data.name); // Bob
```

Always stringify when saving, parse when reading.

Pro Tip

Use localStorage for non-sensitive and persistent data only.

- Preferences ✓
- Cart ✓
- Tokens ✗
- Passwords ✗

sessionStorage

Store data for the duration of a browser session.
Temporary, isolated and session-scoped.

- **Temporary Data** — Data lives only for the session
- **Browser Session** — Cleared when tab/window is closed
- **Session Scoped** — Accessible in the same tab/window
- **JSON Support** — Store & retrieve objects easily
- **Secure** — Data never leaves the browser
- **Fast & Simple** — Lightweight and easy to use

1 What is sessionStorage?

sessionStorage is a Web Storage API that stores key-value pairs for a single browser session.

Data is cleared when the tab or browser is closed.

- Perfect for temporary data
- Survives page reloads in the same tab
- Not accessible from other tabs/windows

2 setItem()

Store data in sessionStorage.

```
// Store string
sessionStorage.setItem('username', 'john');

// Store number (as string)
sessionStorage.setItem('age', '25');

// Store object (use JSON)
const user = { id: 1, name: 'John' };
sessionStorage.setItem('user',
JSON.stringfiy(user));
```

Only strings are stored in sessionStorage. Use JSON.*stringfy*() for objects.

3 getItem()

Retrieve data from sessionStorage.

```
// Get string
const username = sessionStorage.getItem('username');
console.log(usernamio); // john

// Get number (convert if needed)
const age = sessionStorage.getItem('age');
console.log(Number(aIpe)); // 25

// Get object
const user = JSON.parse(sessionStorage.getItem('user'));
console.log(user.name); // John
```

Returns *null* if the key does not exist.

4 removeItem()

Remove a specific item from sessionStorage.

```
// Remove item
sessionStorage.renoveItem('username');

// Check if removed
consolo.log(sessionStorage.getItem('username'));
// null
```

Use the exact key to remove the item.

5 clear()

Remove all items from sessionStorage.

```
// Clear all data
lessionStorage.clear();

consolo.log(sessionStorage.length); // 0
```

Use with caution! This clears all session data.

6 Check & Length

Check keys and total items.

```
// Check if key exists
const hasUser = sessionStorage.getItem('user') !== null;;

// Get all keys
const keys = Object.keys(sessionStorage);

// Get total items
const total = sessionStorage.length;
console.log(keys, total);
```

sessionStorage.length returns the number of items.

7 Example: Save Session State

Save and restore state during a session.

```
// Save state
sessionStorage.setItem('step', '2');

// Restore state
const step = sessionStorage.getItem('step');
if (step) {
    consolo.log('Restored step:', step);
}
```

Useful for wizards, forms, filters, UI state, etc.

8 Use Cases

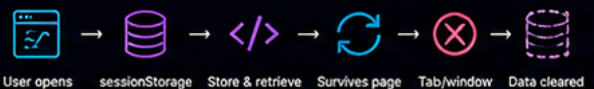
- Form progress (multi-step forms)
- Shopping cart (temporary)
- Filter & search preferences
- UI state (tabs, accordions, etc.)
- Game session data

9 Important Notes

- Data is session-scoped, not persistent.
- Cleared when the tab/window is closed.
- Not shared between different tabs/windows.
- Use JSON for objects/arrays.
- Do not store sensitive data.

sessionStorage Lifecycle

How sessionStorage Works



Data exists only during the page session in that tab/window.

Common Methods

Method	Description	Example
setItem(key, value)	Store data	sessionStorage.setItem('key', 'value')
getItem(key)	Get data	sessionStorage.getItem('key')
removeItem(key)	Remove data	sessionStorage.renoveItem('key')
clear()	Clear all data	sessionStorage.clear()
key(index)	Get key by index	sessionStorage.key(0)
length	Get total items	sessionStorage.length

Store Objects with JSON

```
// Save object
const cart = { id: 101, items: ['pen', 'book'] };
sessionStorage.setItem('cart', JSON.stringfy(cart));

// Load object
const cartdata = JSON.parse(
sessionStorage.getItem('cart') );
console.log(cartdata.items); // ['pen', 'booh']
```

Always stringify when saving, parse when reading.

Pro Tips

- Great for temporary UI state & user session flow.
- Use localStorage if you need data to persist.
- Avoid storing large amounts of data.
- Perfect for multi-step forms, wizards, and carts.
- Remember: Session = Tab/Window lifetime.

Summary

sessionStorage is ideal for temporary, session-based data that should not persist after the tab or browser is closed.

Performance Optimization

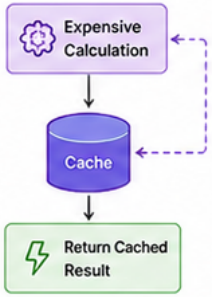
Optimize your React applications for better performance and a fast UI.

Memoization	React.memo	useMemo	useCallback	Lazy Loading	Code Splitting	Optimization	Fast UI
Avoid unnecessary re-renders	Prevent re-render of function components	Memoize expensive calculations	Memoize functions to prevent re-renders	Load components on demand	Split code into smaller chunks	Improve speed, UX and score	Smoother, faster, better experience

1 Memoization – Why?

Memoization stores the result of expensive operations and returns the cached result when the same inputs occur again.

- Reduces re-renders
- Improves performance
- Saves CPU & memory
- Better user experience



2 React.memo

Wrap functional components with React.memo to prevent re-render when props don't change.

```
const UserCard = React.memo(({ user }) => {
  console.log('UserCard rendered');
  return (
    <div className="card">
      <h3>{user.name}</h3>
      <p>{user.email}</p>
    </div>
  );
});
```

Re-renders only when props change.

3 useMemo

Memoize expensive calculations and return the cached value.

```
const total = useMemo(() => {
  console.log('Calculating total...');
  return items.reduce((sum, item) => sum + item.price, 0);
}, [items]); // Recalculate only when items change
```

Use for expensive calculations.
Add dependencies array.

4 useCallback

Memoize callback functions to avoid passing a new function on every render.

```
const handleClick = useCallback((id) => {
  console.log('Item clicked:', id);
  // do something
}, [dependencies]); // Function changes only
                    // when dependencies change
```

Useful when passing callbacks to memoized child components.

5 Lazy Loading

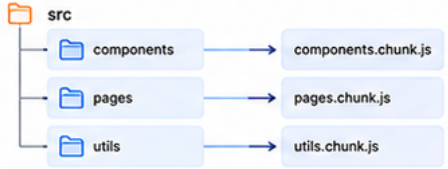
Load components only when they are needed.

```
const About = React.lazy(() => import('./About'));

<Suspense fallback={<div>Loading...</div>}>
  <About />
</Suspense>
```

Improves initial load time.

6 Code Splitting

Split your code into smaller bundles.
Load only what's needed.



Smaller bundles = Faster load.

7 Other Optimization Tips

- Avoid inline functions in JSX
- Use keys properly in lists
- Avoid unnecessary re-renders
- Debounce / Throttle events
- Optimize images and assets
- Use production build
- Remove console.logs in production

Small changes = Big performance boost!

8 Performance Checklist

- Use React.memo for components
- Use useMemo for expensive values
- Use useCallback for functions
- Implement lazy loading
- Apply code splitting
- Optimize images & assets
- Measure performance
- Remove unused dependencies

Before vs After

Before Optimization	After Optimization
Slow load time	Faster load time
High re-renders	Minimal re-renders
Large bundle size	Smaller bundle size
Poor user experience	Smooth & fast UI

Measure Performance

Lighthouse → React DevTools → Profiler → Analyze → Optimize

Measure → Analyze → Optimize → Repeat

Recommended Tools

React DevTools → Lighthouse → Web Vitals → Bundle Analyzer → Why Did You Render

Use the right tools to build high performance apps.

Pro Tip

Performance is not about writing less code, it's about writing the right code.

Fast UI = Happy Users

SEO in React

Make your React apps search engine friendly and rank higher!

- **Meta Tags** — Define how search engines see your page
- **React Helmet** — Manage head content in React
- **Title** — Set unique & relevant page titles
- **Description** — Write clear meta descriptions
- **Open Graph** — Improve social media sharing
- **Structured Data** — Help search engines understand content
- **SSR** — Server Side Rendering for better SEO

1 Why SEO Matters in React?

React is a client-side framework.
Search engines may not see your content without extra setup.

Without SEO	With SEO
Empty / Limited Content	Full Content Indexed

- Better search engine visibility
- More organic traffic
- Higher ranking & user trust

2 Install React Helmet

The easiest way to manage meta tags in React.

```
# Install
npm install react-helmet-async

# or
yarn add react-helmet-async
```

React Helmet
Manage your document head easily.

Use react-helmet-async for server-side rendering support.

3 Basic Meta Tags with React Helmet

Set title, description and other meta tags.

```
import { Helmet } from 'react-helmet-async';

function Home() {
  return (
    <>
     <Helmet>
      <title>Awesome App | Home</title>
      <meta name="description" content="Best app
        for productivity and learning." />
      <meta name="keywords" content="react, seo,
        javascript, web development" />
     </Helmet>
     <h1>Welcome to Awesome App</h1>
} </>
```

Each page should have a unique title & description.

4 Open Graph (Social Sharing)

Control how your page looks when shared.

```
<Helmet>
  <meta property="og:title" content="Awesome App" />
  <meta property="og:description" content="Best app
    for productivity and learning." />
  <meta property="og:image" content="https://yourapp.com/
    og-image.jpg" />
  <meta property="og:url" content="https://yourapp.com/" />
  <meta property="og:type" content="website" />
</Helmet>
```

Improves how your content appears on Facebook, Twitter, LinkedIn and other platforms.

5 Structured Data (JSON-LD)

Help search engines understand your content better.

```
<Helmet>
  <script type="application/ld+json">
  {
    "@context": "https://schema.org",
    "@type": "WebSite",
    "name": "Awesome App",
    "url": "https://yourapp.com",
    "description": "Best app for productivity
     and learning."
  }
  </script>
</Helmet>
```

Boosts rich results and improves click-through rate.

6 SSR (Server Side Rendering)

Search engines can easily crawl server-rendered pages.

Server Render → HTML Ready for Bots → Search Engine Bot

- Faster first contentful paint
- Better SEO & social sharing
- Use Next.js for easiest SSR in React

7 Dynamic Page SEO Example

SEO for dynamic routes.

```
import { Helmet } from 'react-helmet-async';
function BlogPost({ post }) {
  return (
    <Helmet>
      <title>{post.title} | Awesome App</title>
      <meta name="description" content={post.excerpt} />
      <meta property="og:title" content={post.title} />
      <meta property="og:description" content={post.excerpt} />
      <meta property="og:image" content={post.image} />
      <meta property="og:url" content={`https://yourapp.com/
        blog/${post.slug}`} />
    </Helmet>
} );
```

Always generate unique meta tags for each page.

8 SEO Best Practices

- Use meaningful & unique titles
- Write clear meta descriptions
- Add Open Graph & Twitter Card tags
- Use structured data (JSON-LD)
- Enable SSR / Prerendering
- Improve page speed
- Mobile friendly & good UX
- Use semantic HTML

SEO Workflow in React

Build React Page → Add Meta Tags with Helmet → Add Structured Data → Render (SSR / Prerender) → Search Engines Index Page

Better SEO → More Traffic → More Users → More Growth

React Helmet Common Tags

Tag	Purpose	Example
<title>	Page title	<title>About Us | App</title>
<meta name="description" />	Page description	<meta name="description" content="Best app" />
<meta name="keywords" />	Keywords (less important now)	<meta name="keywords" content="react, seo" />
<meta property="og:*" />	Social sharing (Open Graph)	<meta property="og:title" content="App" />
<script type="application/ld+json">	Structured data	<script type="application/ld+json">{...}</script>
<link rel="canonical" href="...">	Canonical URL	<link rel="canonical" href="https://app.com" />
<meta name="robots" content="...">	Indexing control	<meta name="robots" content="index,follow" />

Tools to Test SEO

- **Google Search Console** — Check indexing & performance
- **Google Rich Results Test** — Test structured data
- **PageSpeed Insights** — Analyze speed & performance
- **Lighthouse** — Audit SEO, performance, accessibility

Pro Tip

For best SEO in React, use **Next.js** (SSR + Automatic optimizations) and focus on content quality & user experience.

Next.js Introduction

The React Framework for Production 🚀

Build fast, SEO-friendly, and scalable web apps.

SSR & SSG — Pre-rendered for better performance

App & Pages Router — Flexible routing for every need

API Routes — Build backend inside your app

Image Optimization — Automatic, responsive and modern formats

SEO Friendly — Better rankings out of the box

Fast & Scalable — Optimized by default for speed

1 What is Next.js?

Next.js is a React framework that enables features like Server Side Rendering, Static Site Generation, file-based routing, API routes, image optimization and more.

- Built by Vercel
- Production-ready
- Great Developer Experience
- SEO & Performance Focused

2 Rendering Methods

SSR (Server Side Rendering)

- Page generated on each request
- Always up-to-date
- Better for dynamic content

SSG (Static Site Generation)

- Page generated at build time
- Super fast
- Better for content that changes infrequently

3 Routing in Next.js

Pages Router (Traditional)

```
/pages
  index.js      → /
  about.js      → /about
  blog
    [id].js     → /blog/:id
  _app.js       → (Global Layout)
```

Simple and familiar file-based routing.

App Router (New)

```
/app
  layout.js     → (Root Layout)
  page.js       → /
  about
    page.js     → /about
  blog
    [id]
      page.js   → /blog/:id
```

Built for modern React (Server Components, Streaming, Nested Layouts).

4 API Routes

Create backend APIs inside your Next.js app.

```
// pages/api/hello.js
export default function handler(req, res) {
  if (req.method === 'GET') {
    res.status(200).json({
      message: 'Hello from Next.js API!'
    });
  } else {
    res.status(405).json({ message: 'Method Not Allowed' })
  }
}
```

- Serverless by default (Vercel)
- Supports GET, POST, PUT, DELETE, etc.
- Easy to integrate with frontend

5 Image Optimization

Next.js <Image> component automatically optimizes images.

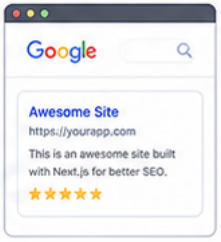
- Automatic resizing
- Modern formats (WebP, AVIF)
- Lazy loading by default
- Prevent layout shift

```
import Image from 'next/image'

export default function Home() {
  return (
    <Image
      src="/hero.jpg"
      alt="Hero Image"
      width={1200}
      height={600}
      priority
    />
  )
}
```

Better performance and Core Web Vitals.

6 SEO Benefits

- SSR & SSG for better search engine visibility
- Automatic meta tags with next/head
- Fast page loads
- Clean URLs
- Open Graph & Twitter Cards support



Better Rankings, More Traffic 📈

7 Next.js Features at a Glance

- **Blazing Fast** — Optimized by default
- **Fullstack** — Frontend + Backend in one framework
- **File-based Routing** — Zero config
- **Built-in Security** — CSRF, Headers & more
- **Data Fetching** — getServerSideProps, getStaticProps, Server Components
- **Middleware** — Edge Runtime & Middleware support
- **Easy Deploy** — One-click deploy on Vercel
- **Huge Ecosystem** — Active & growing community

Everything you need to build modern web apps! 🚀

8 When to Use Next.js?

- Marketing Websites (SEO & Performance)
- E-commerce Stores
- Blogs & Content Sites
- Dashboards & SaaS Apps
- Web Applications with API
- Any React Project that needs More Power!

From simple sites to complex apps – Next.js fits all! ⭐

Rendering Comparison

Feature	SSR (Server Side)	SSG (Static Generation)
Render Time	On each request	At build time
Data	Always fresh	May be stale (until rebuild)
Performance	Good	Excellent
SEO	Excellent	Excellent
Use Case	Dynamic, user-specific	Blogs, docs, marketing

Folder Structure (Overview)

Pages Router

```
/pages
  /api
    index.js
    _app.js
    about.js
    blog/[id].js
```

App Router

```
/app
  /api
    layout.js
    page.js
    about/page.js
    blog/[id]/page.js
```

Key Config File

```
// next.config.js
/** @type {import('next').NextConfig} */
const nextConfig = {
  reactStrictMode: true,
  images: {
    formats: ['image/avif', 'image/webp']
  },
}
module.exports = nextConfig
```

Get Started

Next.js + React = Powerful Combination ❤️

Testing in React

Write tests with confidence. Ship bug-free code.

- **Jest** — Delightful JavaScript Testing Framework
- **React Testing Library** — Test components the way users use them
- **Unit Test** — Test individual functions / components
- **Integration Test** — Test how multiple parts work together
- **Mock** — Isolate dependencies with mocks & stubs
- **Assertions** — Verify expected results
- **Test Cases** — Define scenarios to validate behavior
- **Coverage** — Measure test coverage and improve quality

1 Why Testing?

Tests help you catch bugs early, refactor safely, and document behavior.

- Confidence in your code
- Prevents regressions
- Better design & maintainability
- Happy users 🙂

2 Jest – Test Runner

Jest runs your tests, provides assertions, mocks, and snapshots out of the box.

```
// sum.test.js
function sum(a, b) {
  return a + b;
}

test('adds 1 + 2 to equal 3', () => {
  expect(sum(1, 2)).toBe(3);
});
```

Run tests: `npx jest`

3 React Testing Library

Test components from the user's perspective. Encourages better testing practices.

```
// Button.test.jsx
import { render, screen } from '@testing-library/react';
import Button from './Button';

test('renders button text', () => {
  render(<Button>Click Me</Button>);
  expect(screen.getByRole('button', { name: /click me/i }))
    .toBeInTheDocument();
});
```

Guiding principle: The more your tests resemble the way your software is used, the more confidence they can give you.

4 Unit Test

Test a single unit: function, hook, or component in isolation.

```
// counter.test.jsx
import { renderHook, act } from '@testing-library/react';
import useCounter from './useCounter';

test('increments counter', () => {
  const { result } = renderHook(() => useCounter());
  act(() => result.current.increment());
  expect(result.current.count).toBe(1);
});
```

Fast, reliable and easy to pinpoint failures.

4 Integration Test

Test how multiple parts work together. (Component + API + Router, etc.)

```
// user-list.test.jsx
import { render, screen, waitFor } from
  '@testing-library/react';
import UserList from './UserList';

test('fetches and displays users', async () => {
  render(<UserList />);
  expect(screen.getByText(/loading/i)).toBeInTheDocument();
  await waitFor(() =>
    expect(screen.getByText('John Doe')).toBeInTheDocument()
  );
});
```

Closer to real user flows.

6 Mocking Dependencies

Use mocks to isolate the unit under test.

```
// api.js
export const fetchUser = () => fetch('/api/user');

// api.test.js
import { fetchUser } from './api';
global.fetch = jest.fn(() =>
  Promise.resolve({ json: () => Promise.resolve({ id: 1 }) })
);

test('fetchUser returns user', async () => {
  const user = await fetchUser();
  expect(user.id).toBe(1);
  expect(fetch).toHaveBeenCalledWith('/api/user');
});
```

`jest.fn()` helps create mock functions.

7 Assertions

Verify expected results with rich matchers.

- `expect(value).toBe(expected)` — Strict equality
- `expect(value).toEqual(expected)` — Deep equality
- `expect(value).toBeTruthy()` — Truthiness
- `expect(array).toContain(item)` — Array contains item
- `expect(fn).toThrow(error)` — Function throws error

Good assertions make great tests!

8 Test Cases (AAA Pattern)

Arrange → Act → Assert

Arrange	Set up everything your test needs.
Act	Trigger the behavior you want to test.
Assert	Verify the outcome with assertions.

Keep tests small, focused and independent.

9 Test Coverage

Measure how much of your code is tested.

Coverage Report	
Statements	92.4%
Branches	87.1%
Functions	93.2%
Lines	91.0%
Uncovered Lines	23

- High coverage means fewer surprises.
- Aim for critical paths, not 100% blindly.

Run: `npx jest --coverage`

10 Helpful Commands

`npx jest`	Run all tests
`npx jest --watch`	Watch mode
`npx jest --coverage`	Generate coverage
`npx jest -t "name"`	Run tests by name
`npx jest path/to/file.test.js`	Run specific test file
`npx jest --updateSnapshot`	Update snapshots

Fast feedback loop = productive development!

Typical Project Structure

```
src/
  components/
    Button.jsx
    Button.test.jsx
  hooks/
    useCounter.js
    useCounter.test.js
  utils/
    sum.js
    sum.test.js
```

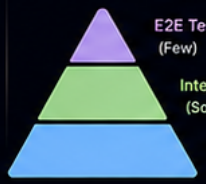
Best Practices

- Test behavior, not implementation
- Use meaningful test names
- Keep tests independent
- Avoid testing third-party libraries
- Refactor tests as you refactor code

Example – Component Test

```
// Welcome.test.jsx
import { render, screen } from '@testing-library/react';
import Welcome from './Welcome';

test('renders welcome message', () => {
  render(<Welcome name="React Dev" />);
  expect(screen.getByText(/welcome, react dev/i)).toBeInTheDocument();
});
```

Test Pyramid



Pro Tip

Tests are not a cost, they are an investment. Ship with confidence and iterate faster! 🚀

Deployment

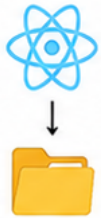
Take your React app live for the world! 🌎

Build — Create optimized production build → **Deploy** — Upload files to hosting platform → **Host** — Serve your app worldwide → **Secure** — HTTPS, Headers & Best practices → **Fast** — CDN, Caching & Performance → **Scale** — Handle traffic & grow your app

1 Build Your App

Create an optimized production build for the best performance.

```
# Install dependencies
npm install

# Create production build
npm run build

# Output folder: /build
# (or /dist if using Vite)
```



Optimized, minified and ready for production!

2 Vercel

The platform built for Next.js & modern frontend apps.

1. Push your code to GitHub
2. Import your repository on Vercel
3. Vercel auto-detects framework
4. Click Deploy
5. Your app is live! 🎉

Live URL: https://your-app.vercel.app

Automatic Deploys • HTTPS • CDN
Global Edge Network • Preview Deploys

3 Netlify

All-in-one platform for hosting modern web projects.

1. Connect your GitHub repository
2. Build command: `npm run build`
3. Publish directory: `build`
4. Click Deploy site
5. Your app is live! 🚀

Live URL: https://your-app.netlify.app

Auto Deploys • Form Handling • Functions
HTTPS • Global CDN • Easy Rollbacks

4 GitHub Pages

Free hosting directly from your GitHub repository.

1. Go to Settings > Pages
2. Select branch (`main` / `gh-pages`)
3. Select folder (`/build`)
4. Save and wait a few minutes
5. Your site is live! 🎉

Live URL: https://username.github.io/repo

Free • Easy to set up • HTTPS
Great for docs & static sites

5 Firebase Hosting

Fast and secure hosting by Google with powerful features.

1. Install Firebase CLI
   `npm install -g firebase-tools`
2. Login to Firebase
   `firebase login`
3. Initialize Hosting
   `firebase init hosting`
4. Deploy your app
   `firebase deploy`

Blazing Fast • HTTPS • CDN • Rewrites
Easy Rollbacks • Preview Channels • More

Hosting Platforms Comparison

Platform	Free Plan	Custom Domain	CI/CD	Best For
Vercel	✓ Yes	✓ Yes	✓ Yes	Next.js, React Apps
Netlify	✓ Yes	✓ Yes	✓ Yes	Static Sites, JAMStack
GitHub Pages	✓ Yes	✓ Yes	–	Docs, Portfolios
Firebase Hosting	✓ Yes	✓ Yes	✓ Yes	Web Apps, SPAs
AWS Amplify	✓ Yes	✓ Yes	✓ Yes	Fullstack Apps

General Deployment Steps

1. Build
Create an optimized production build.

2. Upload
Upload the build folder to your hosting platform.

3. Configure
Set rewrites, redirects, and environment variables if needed.

4. Go Live
Get your live URL and share it with the world! 🚀

Pro Tips

- Always test your build locally (`npm run build`).
- Use environment variables for API keys.
- Enable HTTPS for security.
- Set up redirects for SPA (`/* → /index.html`).
- Use CDN for better performance.
- Monitor your app after deployment.

Clean Build + Good Hosting = Happy Users 😍

Example: Deploy to Vercel

```
# 1. Push code to GitHub
 git add .
 git commit -m "Ready to deploy"
 git push origin main
|
# 2. Import repo on Vercel and click Deploy
|
# 3. Done! 🎉
```

Your app is now live and accessible worldwide! 🌎
Keep building and keep shipping! 🚀

React Folder Structure

A scalable and maintainable project structure for React applications 🚀

Organized — Well structured and easy to navigate

Scalable — Easy to grow your application

Maintainable — Clean code and separation of concerns

Production Ready — Best practices for real-world apps

```
my-react-app/
├── public/
│   ├── index.html
│   └── favicon.ico
├── src/
│   ├── assets/
│   │   ├── logo.png
│   │   └── styles/
│   │       └── global.css
│   ├── components/
│   │   ├── Button/
│   │   ├── Card/
│   │   └── index.js
│   ├── pages/
│   │   ├── Home/
│   │   ├── About/
│   │   └── index.js
│   ├── layouts/
│   │   └── MainLayout.jsx
│   ├── hooks/
│   │   ├── useAuth.js
│   │   └── useWindowSize.js
│   ├── context/
│   │   └── AuthContext.js
│   ├── utils/
│   │   ├── api.js
│   │   └── helpers.js
│   ├── App.jsx
│   └── main.jsx
├── .env
├── .gitignore
├── package.json
└── README.md
```

src/

Contains all the source code of the application.

components/

Reusable UI components.

```
Button.jsx
Card.jsx
```

pages/

Page-level components used for routing.

```
Home.jsx
About.jsx
```

layouts/

Layout components that wrap pages (Header, Sidebar, Footer etc.).

```
MainLayout.jsx
AuthLayout.jsx
```

assets/

Static assets like images, icons, fonts and global styles.

```
logo.png
global.css
```

hooks/

Custom React hooks for reusable logic.

```
useAuth.js
useWindowSize.js
```

context/

React Context API for global state management.

```
AuthContext.js
ThemeContext.js
```

utils/

Utility functions, API calls, helpers and constants.

```
api.js
helpers.js
```

App.jsx & main.jsx

Root component and app entry point.

```
App.jsx
main.jsx
```

Structure Benefits

- Easy to navigate and understand
- Separation of concerns
- Reusability of components and hooks
- Scalable for small to large applications
- Better collaboration in teams

Quick Tips

Feature-Based Components
Keep related components together in feature folders.

Use Index Files
Export components and modules using index.js for clean imports.

Alias for Imports
Use path aliases (e.g. @components) to avoid long relative paths.

Environment Variables
Store API keys and secrets in .env files. Never commit them!

Example Import with Alias

```
import Button from '@components/Button';  // instead of  import Button from '../../components/Button';
```

Configure aliases in jsconfig.json or tsconfig.json for cleaner imports.

React Best Practices

Write clean, scalable and maintainable React applications 🚀

Maintainable — Easy to update and extend

Performant — Better user experience

Scalable — Grow from small to large apps

Fewer Bugs — Clean code means fewer issues

Developer Friendly — Easy to read and collaborate

1 Reusable Components

Build small, reusable and composable components.

- Keep components small and focused
- Pass props, not global data
- Reuse across the app

```jsx
// Button.jsx
const Button = ({
  text, onClick,
  type = 'button'
}) => {
  return (
    <button
      type={type}
      onClick={onClick}
      className="btn">{text}
    </button>
  );
};
export default Button;
```

2 Clean Code

Write readable and self-explanatory code.

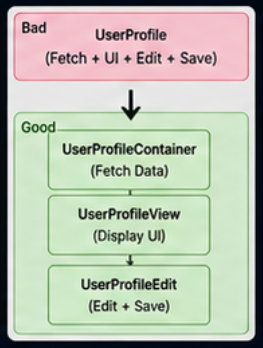
- Use meaningful names
- Keep functions short
- Remove unused code
- Add helpful comments

```jsx
// Good
const calculateTotal = (items) => {
  return items.reduce((sum, item) =>
    sum + item.price * item.qty, 0)
  );
};

// Avoid
const calc = (i) => i.reduce(
  (s, x) => s + x.p * x.q, 0)
);
```

3 Single Responsibility

A component should have one job and do it well.

- One component, one responsibility
- Break large components into smaller ones
- Easier to test and maintain



4 Custom Hooks

Extract and reuse stateful logic using Custom Hooks.

- Share logic between components
- Keep components clean
- Follow naming: useXxx

```jsx
// useCounter.js
import { useState } from 'react';

export const useCounter = (initial
= 0) => {
  const [count, setCount] =
    useState(initial);
  const increment = () =>
    setCount(c => c + 1);
  return { count, increment,
           setCount };
};
```

5 Folder Structure

Organize files in a scalable and predictable way.

- Feature-based structure
- Keep components, hooks, utils separate
- Easy to navigate

```
src/
  components/
    Button.jsx
    Card.jsx
  pages/
    Home.jsx
    About.jsx
  hooks/
    useAuth.js
    useCounter.js
  utils/
    api.js
  App.js
  main.jsx
```

6 Naming Convention

Use consistent and meaningful names across the project.

Abc

- Components: PascalCase
- Hooks: camelCase (useXxx)
- Files: PascalCase for components, camelCase for hooks & utils
- Constants: UPPER_SNAKE_CASE
- Boolean: is, has, can, should

✓ UserProfile.jsx
✓ useFetchData.js
✓ isLoggedIn
✓ MAX_RETRY_COUNT

✗ userprofile.jsx
✗ Usefetchdata.js
✗ data-util.js

7 Performance

Optimize your app for the best performance.

- Use React.memo
- Use useMemo & useCallback
- Avoid unnecessary re-renders
- Virtualize long lists
- Optimize images and assets

```jsx
const ExpensiveComponent =
 React.memo(({ data }) => {
  const processed = useMemo(() =>
  heavyComputation(data), [data]);
  return <div>{processed}</div>;
});

const handleClick = useCallback(
  () => { /* do something */ },
  []
);
```

8 Code Splitting

Split your code and load only what's needed.

- Use React.lazy & Suspense
- Split by routes or features
- Improves initial load time

```jsx
// Lazy load component
const Dashboard = React.lazy(() =>
 import('./pages/Dashboard'));

function App() {
 return (
  <Suspense fallback={<Loader/>}>
   <Routes>
    <Route path="/dashboard"
     element={<Dashboard />} />
   </Routes>
  </Suspense>
 );
}
```

Pro Tips

Keep It Simple — Don't over-engineer. Start simple.

Test Your Code — Write unit and integration tests for confidence.

Keep Dependencies Updated — Regular updates for security and performance.

Code Reviews — Review code, share knowledge, improve quality.

Write Documentation — Document components, hooks and complex logic.

Measure & Improve — Use tools like React DevTools and Lighthouse.

React Interview Questions

Most Asked React.js Interview Questions 🚀

Commonly Asked — Frequently asked in React interviews

Concept Based — Focus on core concepts

Job Ready — Prepare better, crack interviews

Career Boost — Stronger concepts, better opportunities

1 What is Virtual DOM?

Q: What is Virtual DOM in React?

A: Virtual DOM is a lightweight JavaScript representation of the actual DOM. React updates the Virtual DOM first, compares the changes (diffing) and then updates only what's changed in the real DOM.

2 What is JSX?

Q: What is JSX?

A: JSX is a syntax extension for JavaScript. It allows us to write HTML-like code in JavaScript and makes React code more readable and expressive.

```
const element = (
  <h1 className="title">
    Hello, React!
  </H1>
);
```

3 Props vs State?

Q: What is the difference between Props and State?

A:

Props	State
Passed to component	Managed within the component
Immutable (read-only)	Mutable (can change)
From parent	From component itself
Used for configuration	Used for dynamic data

4 What are Hooks?

Q: What are Hooks in React?

A: Hooks are functions that let you use state and other React features without writing a class.

Common Hooks:

- useState() – State management
- useEffect() – Side effects
- useContext() – Access context
- useRef() – Reference values

Hooks can only be used in functional components.

5 What is Lifecycle?

Q: What are the lifecycle methods in React (Class Components)?

A:

Mounting
`constructor() → render() → componentDidMount()`

Updating
`render() → componentDidUpdate()`

Unmounting
`componentWillUnmount()`

(In functional components, we use useEffect hook)

6 Why are Keys Important?

Q: Why do we need keys in lists?

A: Keys help React identify which items have changed, are added, or removed. They improve performance and help maintain state.

Good ✅

```
{items.map(item => (
  <li key={item.id}>{item.name}</li>
))}
```

Bad ❌

```
{items.map((item, index) => (
  <li key={index}>{item.name}</li>
))}
```

7 What is Context API?

Q: What is Context API?

A: Context API is used to manage global data (like theme, auth) without prop drilling.

```
const ThemeContext =
  React.createContext();

<ThemeContect.Provider
  value={theme}>
  {children}
</ThemeContext.Provider>

const theme =
useContext(ThemeContext);
```

8 What is Redux?

Q: What is Redux?

A: Redux is a state management library for JavaScript apps.
It follows a predictable pattern.

Core Concepts:

- Store – holds the state
- Action – describes changes
- Reducer – updates the state
- Dispatch – triggers actions
- UseSelector – reads state
- UseDispatch – dispatches actions

Popular with complex state in large apps.

1 What is Virtual DOM?

Q: What is Virtual DOM in React?

A: Virtual DOM is a lightweight JavaScript representation of the actual DOM. React updates the Virtual DOM first, compares the changes (diffing) and then updates only what's changed in the real DOM.

2 What is JSX?

Q: What is JSX?

A: JSX is a syntax extension for JavaScript. It allows us to write HTML-like code in JavaScript and makes React code more readable and expressive.

```
const element = (
  <h1 className="title">
    Hello, React!
  </H1>
);
```

3 Props vs State?

Q: What is the difference between Props and State?

A:

Props	State
Passed to component	Managed within the component
Immutable (read-only)	Mutable (can change)
From parent	From component itself
Used for configuration	Used for dynamic data

4 What are Hooks?

Q: What are Hooks in React?

A: Hooks are functions that let you use state and other React features without writing a class.

Common Hooks:

- useState() – State management
- useEffect() – Side effects
- useContext() – Access context
- useRef() – Reference values

Hooks can only be used in functional components.

5 What is Lifecycle?

Q: What are the lifecycle methods in React (Class Components)?

A:

Mounting
`constructor() → render() → componentDidMount()`

Updating
`render() → componentDidUpdate()`

Unmounting
`componentWillUnmount()`

(In functional components, we use useEffect hook)

6 Why are Keys Important?

Q: Why do we need keys in lists?

A: Keys help React identify which items have changed, are added, or removed. They improve performance and help maintain state.

Good ✅

```
{items.map(item => (
  <li key={item.id}>{item.name}</li>
))}
```

Bad ❌

```
{items.map((item, index) => (
  <li key={index}>{item.name}</li>
))}
```

7 What is Context API?

Q: What is Context API?

A: Context API is used to manage global data (like theme, auth) without prop drilling.

```
const ThemeContext =
  React.createContext();

<ThemeContext.Provider
  value={theme}>
  {children}
</ThemeContext.Provider>

const theme =
useContext(ThemeContext);
```

8 What is Redux?

Q: What is Redux?

A: Redux is a state management library for JavaScript apps.
It follows a predictable pattern.

Core Concepts:

- Store – holds the state
- Action – describes changes
- Reducer – updates the state
- Dispatch – triggers actions
- UseSelector – reads state
- UseDispatch – dispatches actions

Popular with complex state in large apps.

Pro Tips

Understand the WHY — Focus on concepts, not just definitions.

Write Code — Practice by building small projects.

Stay Updated — React is evolving, keep learning.

Practice More — Solve interview Qs and coding challenges.

Think Out Loud — Explain your approach clearly in interviews.

Build Real Projects — Apply concepts in real-world projects.

Common React Errors

Errors you'll face. Solutions you'll love.

Read the Error — Understand what React is telling you.

Find the Cause — Trace the source of the issue.

Apply the Fix — Use the right solution.

Test & Verify — Confirm the error is resolved.

1 Invalid Hook Call

Hooks can only be called inside of the body of a function component or a custom Hook.

Problem

```
function MyComponent() {
  if (true) {
    const [count, setCount] = useState(0);
  }
  return <div>{count}</div>;
}
```

Fix

- Only call Hooks at the top level.
- Don't call Hooks inside loops, conditions, or nested functions.

2 Missing Key

Each child in a list should have a unique "key" prop.

Problem

```
{items.map((item, index) =>
  <li>{item.name}</li>
)}
```

Fix

- Use a stable and unique key.

```
{items.map(item =>
  <li key={item.id}>{item.name}</li>
)}
```

3 Undefined Props

Accessing properties of undefined or null causes runtime errors.

Problem

```
function User({ user }) {
  return <div>{user.name}</div>;
}
// user is undefined
```

Fix

- Use optional chaining or default values.

```
return <div>{user?.name || "Guest"}</div>;
```

4 Infinite Loop

State updates inside useEffect without proper dependencies.

Problem

```
useEffect(() => {
  setCount(count + 1);
}); // Missing dependency array
```

Fix

- Add dependency array or correct dependencies.

```
useEffect(() => {
  setCount(c => c + 1);
}, []);
```

5 Module Not Found

React can't find the module you're trying to import.

Problem

```
Module not found: Can't resolve
'./components/Button'
```

Fix

- Check the file path and file name.
- Verify the file exists.
- Restart the dev server.

6 Hydration Error

Server and client render output do not match.

Problem

```
Warning: Text content did not match.
Server: "Hello"
Client: "Hello World"
```

Fix

- Ensure the same data on server and client.
- Avoid using browser-only APIs during SSR.
- Use useEffect for client-only changes.

7 TypeError

Trying to access properties or methods of undefined or null.

Problem

```
TypeError: Cannot read properties
of undefined (reading 'map')
```

Fix

- Check if the value exists before using it.
- Use optional chaining.

```
data?.map(item => item.name)
```

8 Console Errors

Errors shown in the browser console that break functionality.

Problem

```
Uncaught ReferenceError: data is not defined
  at App.js:10:15
Uncaught (in promise) Error: Request failed
  with status code 500
```

Fix

- Read the console message carefully.
- Trace the error to the exact line.
- Fix the issue and test again.

Best Practices

- Read errors carefully
- Understand the root cause
- Fix and test incrementally
- Keep your code clean
- Update dependencies regularly

Quick Debug Tips

Check the stack trace — It shows where the error occurred.

Use console.log() — Log values to debug issues.

Reproduce the issue — Create minimal code to isolate the bug.

Clear cache & restart — Sometimes a simple restart fixes it.

Read the docs — React docs explain many errors well.

Remember

Errors are normal. Fixing them makes you a better developer!

React Project Architecture

A scalable and maintainable architecture for modern React applications. 🚀

Scalable — Easily grow from small to big apps

Maintainable — Clean structure & separation of concerns

Reusable — Reusable components, hooks & services

Testable — Easy testing and better reliability

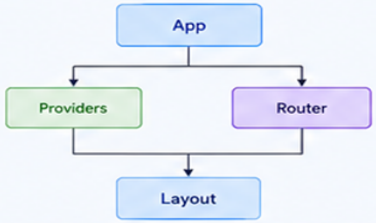
Application Flow

1. **User Request** — User interacts with the application
2. **Routing** — React Router matches the route
3. **Page** — Page component is loaded
4. **Components** — Reusable components are rendered
5. **Data Fetching** — Custom hooks call services / API
6. **UI Update** — State updates and UI re-renders

1. App Component (Root)

Entry point of the application.

```jsx
// App.jsx
function App() {
  return (
    <BrowserRouter>
      <AppProviders>
        <MainLayout />
      </AppProviders>
    </BrowserRouter>
  );
}
```



2. Layouts

Define the common structure like header, sidebar, footer.

```jsx
// MainLayout.jsx
const MainLayout = () => (
  <div className="app-layout">
    <Header />
    <Sideber />
    <main>
      <Outlet />
    </main>
  </Footer />
);
```

3. Pages

Route-level components that represent pages.

```jsx
// pages/Dashboard.jsx
const Dashboard = () => {
  const { data } = useDashboard();
  return (
    <div className="page">
      <h1>Dashboard</h1>
      <Stats data={data} />
    </div>
  );
};
```

4. Components

Reusable UI components used across pages.

```jsx
// components/Stats.jsx
const Stats = ({ data }) => (
  <div className="stats">
    {data.map(item => (
      <StatCard key={item.id}
                item={item} />
    ))}
  </div>
);
```

5. Hooks

Custom hooks for reusable logic and state.

```jsx
// hooks/useDashboard.js
export const useDashboard = () => {
  const [data, setData] =
    useState([]);
  useEffect(() => {
    getDashboardData().then(setData);
  }, []);
  return { data };
};
```

6. Services

Handle business logic, API calls and data transformation.

```jsx
// services/dashboardService.js
import api from "../api/client";
export const getDashboardData = async () => {
  const { data } = await api.get("/dashboard");
  return data;
};
```

7. API Layer

Centralized API client for all HTTP requests.

```jsx
// api/client.js
import axios from "axios";
const api = axios.create({
  baseURL: "https://api.example.com",
  timeout: 10000,
});
export default api;
```

8. Routing

Defines all application routes and navigation.

```jsx
// routes/AppRoutes.jsx
import { Routes, Route } from "react-router-dom";
import Dashboard from "../pages/Dashboard";
export default function AppRoutes() {
  return (
    <Routes>
      <Route path="/" element={<Dashboard />} />
      <Route path="/users" element={<Users />} />
      <Route path="*" element={<NotFound />} />
    </Routes>
  );
}
```

Data Flow

- **Components / Pages** (need data)
- ↓ **Custom Hooks** (manage state & logic)
- ↓ **Services** (business logic)
- ↓ **API Layer** (make HTTP requests)
- ↓ **Backend API** (return data)

Project Structure

```
src/
  components/
    Button.jsx
    StatCard.jsx
  layouts/
    MainLayout.jsx
  pages/
    Dashboard.jsx
    Users.jsx
  hooks/
    useDashboard.js
  services/
    dashboardService.js
    userService.js
  api/
    client.js
  routes/
    AppRoutes.jsx
  context/
    AuthContext.js
  assets/
  App.jsx
  main.jsx
.env
.gitignore
package.json
```

Best Practices

Separation of Concerns — Keep UI, logic, and data separate.

Reusable Everything — Create reusable components, hooks and services.

Single Responsibility — Each module should have one responsibility.

Centralized API — Use a single API client and handle errors globally.

Clean & Consistent Code — Follow naming conventions and code standards.

Testable Architecture — Write unit tests for hooks, utils and components.

Useful React Libraries

Powerful libraries to supercharge your React development ⚡

Boost Productivity — Write less code, build more.

Well Tested — Used by millions, trusted by teams.

Easy Integration — Seamless with React ecosystem.

Actively Maintained — Regular updates & strong community.

1. React Router

Declarative routing for React applications.

- Nested routing
- Dynamic routes
- Navigation guards

```
// App.js
import { BrowserRouter, Routes, Route }
 from 'react-router-dom';

 <BrowserRouter>
   <Routes>
     <Route path="/" element={<Home />} />
     <Route path="/about" element={<About />} />
   </Routes>
 </BrowserRouter>
```

2. Axios

Promise based HTTP client for the browser and Node.js.

- Simple API
- Interceptors
- Request & response handling

```
// api.js
import axios from 'axios';

export const getUsers = () =>
  axios.get('https://api.example.com/users");

export const createUser = (data) =>
  axios.post('/users', data);
```

3. Redux Toolkit

Official, opinionated, and powerful Redux toolset.

- Simplified store setup
- Built-in immer
- Less boilerplate

```
// store/slice.js
import { createSlice } from '@reduxjs/toolkit';

const counterSlice = createSlice({
   name: 'counter',
   initialState: { value: 0 },
   reducers: {
      increment: (state) => { state.value += 1 }
   }
});
```

4. React Hook Form

Performant, flexible and easy forms with validation.

- Uncontrolled inputs
- Built-in validation
- Less re-renders

```
// MyForm.js
import { useForm } from 'react-hook-form';

const { register, handleSubmit } = useForm();

const onSubmit = (data) => console.log(data);
```

5. Framer Motion

Production-ready motion library for React.

- Declarative animations
- Gesture support
- Layout animations

```
// FadeIn.js
import { motion } from 'framer-motion';

export const FadeIn = () => (
  <motion.div
    initial={{ opacity: 0 }}
    animate={{ opacity: 1 }}
    transition={{ duration: 0.5 }}
  />
);
```

6. React Icons

Include popular icons in your React projects easily.

- 100+ icon sets
- Customizable
- Tree-shakable

```
// IconExample.js
import { FaReact, FaHeart } from 'react-icons/fa';

const IconExample = () => (
  <div>
    <FaReact size={32} color="#61DAFB" />
    <FaHeart size={24} color="#E91E63" />
  </div>
);
```

7. TanStack Query

Powerful data fetching, caching and synchronization.

- Auto caching
- Background updates
- Pagination & more

```
// useUsers.js
import { useQuery } from '@tanstack/react-query';

const fetchUsers = () => fetch('/api/users').then(res => res.json());

export const useUsers = () =>
  useQuery({ queryKey: ['users'], queryFn: fetchUsers });
```

8. Chart.js

Simple yet flexible JavaScript charting for React.

- Responsive charts
- Multiple chart types
- Easy customization

```
// ChartExample.js
import { Line } from 'react-chartjs-2';

<Line
  data={data}
  options={{ responsive: true }}
/>
```

Pro Tip — Use these libraries together to build modern, scalable and maintainable React applications faster!

Faster Development — Save time with powerful abstractions.

Better UX — Smooth interactions and beautiful UI.

Clean Code — Write less, do more with better APIs.

Scalable Apps — Handle complexity with ease.

React Developer Tools

Essential tools to build, debug, test and ship amazing React applications 🚀

Faster Development — Build & debug more efficiently

Better Debugging — Find & fix issues quickly

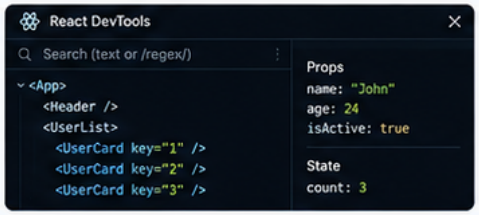
Code Quality — Write clean, consistent and error-free code

Team Collaboration — Work better together with modern tools

1 React DevTools

Official DevTools for React. Inspect components, props, state and performance.

- Component tree
- Props & state inspector
- Hooks inspection
- Highlight updates



2 Chrome DevTools

Powerful in-browser DevTools for debugging and performance analysis.

- Elements & Console
- Network monitoring
- Performance profiling
- Memory & Lighthouse

3 VS Code

Lightweight and powerful code editor with amazing extensions for React development.

- IntelliSense & Autocomplete
- Debugging support
- Integrated Terminal
- Extensions ecosystem

4 ESLint

Find and fix problems in your JavaScript/React code.

- Detect errors
- Enforce code style
- Customizable rules
- Improve code quality

```
// .eslintrc.json
{
  "extends": ["eslint:recommended",
              "plugin:react/recommended"],
  "rules": {
    "no-unused-vars": "warn",
    "react/prop-types": "off"
  }
}
```

5 Prettier

Opinionated code formatter. Keeps your code clean and consistent.

- Auto format code
- Consistent style
- Supports many languages
- Save time & focus

```
// Before
function sum(a,b){return a+b}

// After
function sum(a, b) {
  return a + b;
}
```

6 Vite

Next generation frontend tooling. Blazing fast dev server & optimized builds.

- Instant server start
- Hot Module Replacement
- Optimized build
- ESM based

```
$ npm create vite@latest my-app -- -template react
$ cd my-app
$ npm install
$ npm run dev
  VITE v5.1.0   ready in 196 ms
  → Local:    http://localhost:5173/
```

7 npm

Node Package Manager. Manage dependencies and scripts easily.

- Install packages
- Manage versions
- Run scripts
- Share packages

```
$ npm install axios
$ npm install -D eslint prettier
$ npm run dev
$ npm run build

added 84 packages, and audited 85 packages
found 0 vulnerabilities
```

8 Git

Distributed version control system to track changes in your code.

- Track changes
- Branch & merge
- Collaborate easily
- Version history

```
$ git status
$ git add .
$ git commit -m "feat: add user list"
$ git push origin main
Enumerating objects: 7, done.
Counting objects: 100% (7/7), done.
To github.com:user/repo.git
   a1b2c3d..d4e5f6g  main -> main
```

Pro Tips

Learn Shortcuts — Use shortcuts to speed up your workflow.

Use Extensions — VS Code extensions can boost your productivity.

Keep Updated — Keep your tools updated for the best features & security.

Automate — Use linters, formatters & scripts to automate repetitive tasks.

Work as a Team — Use Git and collaborate effectively with your team.

Test & Debug — Use DevTools and linters to write bug-free React apps.

React Development Workflow

Step-by-step process to build, test and ship awesome React applications. 🚀

Fast — Quick development with hot reload

Scalable — Build maintainable and scalable apps

Reliable — Tested, bug-free and robust

Deploy Anywhere — One build, deploy anywhere

1 Create Project

- Initialize a new React project
- Setup project structure and dependencies
- Configure dev tools

```
# Using Vite (Recommended)
npm create vite@latest
my-react-app -- --template react
cd my-react-app
npm install
npm run dev
```

2 Build Components

- Create reusable UI components
- Use Props for data passing
- Follow component best practices

```
// Button.jsx
export default function Button({
  label, onClick
}) {
  return (
    <button className="btn"
     onClick={onClick}>
      {label}
    </button>
  );
}
```

3 Manage State

- Use useState for local component state
- Use Context or Redux for global state
- Keep state predictable

```
// Counter.jsx
import { useState } from 'react';

export default function Counter() {
 const [count, setCount] =
 useState(0);
 return (
  <button onClick={() => ⟩⟩
   setCount(count + 1)⟩
   Count: {count}
  </button>
 );
}
```

4 Fetch API

- Fetch data from APIs
- Handle loading, error and success states
- Store data in state

```
// useUsers.js
export function useUsers() {
 const [data, setData] =
  useState([]);
 const [loading, setLoading] =
  useState(true);
 useEffect(() => {
  fetch('https://api.example.com/users')
   .then(res => res.json())
   .then(json => setData(json))
   .finally(() => setLoading(false));
 }, []);
 return { data, loading };
}
```

5 Routing

- Setup React Router
- Create routes & nested routes
- Protect routes if needed

```
// App.jsx
import { BrowserRouter,
 Routes, Route } from
 'react-router-dom';

<BrowserRouter>
 <Routes>
  <Route path="/" element={<Home />} />
  <Route path="/about" element={<About />} />
  <Route path="*" element={<NotFound />} />
 </Routes>
</BrowserRouter>
```

6 Testing

- Write unit tests for components
- Test user interactions
- Ensure high coverage and reliability

```
// Button.test.jsx
import { render, screen,
 fireEvent } from
 '@testing-library/react';
import Button from './Button';

test('renders and clicks', () => {
 const handleClick = jest.fn();
 render(<Button label="Click"
  onClick={handleClick} />);
 fireEvent.click(screen.getByText('Click'));
 expect(handleClick).toHaveBeenCalled();
});
```

7 Build

- Create optimized production build
- Run lint and fix issues
- Analyze bundle size

```
# Build for production
npm run build

# Preview production build
npm run preview

# Lint code
npm run lint

# Analyze bundle
npx vite-bundle-visualizer
```

8 Deploy

- Deploy to Vercel, Netlify, or Firebase
- Setup custom domain
- Monitor and measure performance

```
# Deploy to Vercel
npm i -g vercel
vercel
vercel --prod

# Or build & deploy manually
```

Best Practices

Write Clean Code — Follow code standards and naming conventions.

Component Reusability — Build small, reusable components.

Keep It Simple — Avoid over-engineering. Keep components simple.

Performance First — Use memo, lazy loading and code splitting.

Security Matters — Validate inputs and use `.env` for secrets.

Test Everything — Write tests and prevent regressions.

Monitor & Improve — Track performance and improve continuously.

Pro Tip — Use TypeScript, ESLint, Prettier and Husky to maintain high code quality and consistent development.

Recommended Stack: React · Vite · Tailwind CSS · React Query · Zustand / Redux · React Router · Testing Library

React Project Checklist

Everything you need to build a production-ready React application. 🚀

High Quality — Follow best practices and write clean code

Reliable — Handle errors and edge cases

Performant — Optimize for speed and efficiency

Production Ready — Tested, optimized and ready to deploy

1 Responsive UI

Build beautiful UI that works on all devices.

- Use Flexbox / Grid for layouts
- Make UI mobile-first
- Use media queries
- Test on all screen sizes

```
@media (max-width: 768px) {
  .container {
    padding: 1rem;
    flex-direction: column;
  }
}
```

2 Reusable Components

Create reusable and modular components.

- Break UI into small components
- Use Props effectively
- Use children & composition
- Keep components focused

```
function Button({ children, ...props }) {
  return (
    <button className="btn" {...props}>
      {children}
    </button>
  );
}
```

3 Routing

Implement routing for seamless navigation.

- Use React Router
- Create nested routes
- Protect private routes
- Handle 404 pages

```
<Routes>
  <Route path="/" element={<Home />} />
  <Route path="/about" element={<About />} />
  <Route path="/profile" element={<Profile />} />
  <Route path="*" element={<NotFound />} />
</Routes>
```

4 API Integration

Fetch and manage data from APIs efficiently.

- Use Axios / Fetch
- Handle loading states
- Handle API errors
- Use environment variables

```
const getData = async () => {
  try {
    const res = await axios.get('/api/data');
    setData(res.data);
  } catch (error) {
    console.error(error);
  }
};
```

5 Error Handling

Handle errors gracefully and inform users.

- Use Error Boundaries
- Show meaningful messages
- Log errors for debugging
- Handle API & network errors

```
class ErrorBoundary extends React.Component {
  componentDidCatch(error, info) {
    console.error(error, info);
  }
  render() {
    if (this.state.hasError) return <ErrorUI />;
    return this.props.children;
  }
}
```

6 Optimization

Optimize your app for better performance.

- Use React.memo, useMemo, useCallback
- Code splitting
- Lazy load components
- Optimize images & assets

```
const LazyComponent = React.lazy(() =>
  import('./components/LazyComponent');
);

<Suspense fallback={<Loader />}>
  <LazyComponent />
</Suspense>
```

7 Testing

Write tests to ensure your app is bug-free.

- Write unit tests for components
- Test user interactions
- Test API calls (mock)
- Maintain good test coverage

```
import { render, screen } from '@testing-library/react';
import Button from './Button';

test('renders button', () => {
  render(<Button>Click</Button>);
  expect(screen.getByText(/click/i)).toBeInTheDocument();
});
```

8 Deployment Ready

Prepare your app for production deployment.

- Build the production build
- Set environment variables
- Configure meta tags & SEO
- Deploy to hosting platform

```
# Build
npm run build

# Preview
npm run preview

# Deploy (Vercel Example)
vercel --prod
```

Pro Tips

Follow Conventions — Follow consistent naming and folder structure.

Keep It Simple — Write clean, simple and maintainable code.

Keep Dependencies Updated — Update libraries regularly for security and performance.

Monitor Performance — Use tools like Lighthouse and React DevTools.

Get Feedback — Code review and user feedback improve quality.

Documentation — Maintain README and document your code.

React JS Quick Reference

Your cheat sheet for building modern React applications ⚡

Fast — Build UIs faster

Reusable — Component based

Scalable — Build large applications

Reliable — Robust and maintainable

1 JSX

- JSX lets you write HTML in JavaScript.
- It makes UI code more readable.
- Always return a single root element.

```jsx
const element = (
  <div className="container">
    <h1>Hello, React!</h1>
    <p>Welcome to JSX</p>
  </div>
);
export default element;
```

2 Components

- Components are the building blocks.
- Functional components are simple JS functions.
- Use PascalCase for component names.

```jsx
import React from 'react';
function Welcome() {
  return (
    <div>
      <h2>Welcome Component</h2>
    </div>
  );
}
export default Welcome;
```

3 Props

- Props are read-only inputs.
- Pass data from parent to child.
- Access props via function argument.

```jsx
function User({ name, age }) {
  return (
    <div>
      <h3>{name}</h3>
      <p>Age: {age}</p>
    </div>
  );
}

// Usage
<User name="Jeet" age={28} />
```

4 State

- State holds data that can change.
- useState is the most common Hook.
- State updates re-render the component.

```jsx
import { useState } from 'react';
function Counter() {
  const [count, setCount] = useState(0);
  return (
    <div>
      <p>Count: {count}</p>
      <button onClick={() => setCount(count + 1)}>
        Increment
      </button>
    </div>
  );
}
```

5 Hooks

- Hooks let you use state and other React features in functions.
- Common Hooks: useState, useEffect, useContext, useRef.

```jsx
import { useEffect, useState } from 'react';
function Example() {
  const [data, setData] = useState([]);

  useEffect(() => {
    console.log('Component mounted');
    return () => console.log('Cleanup');
  }, []);
  return <div>Hooks Example</div>;
}
```

6 Routing

- React Router enables navigation.
- Use BrowserRouter, Routes, Route.
- Link for navigation without reload.

```jsx
import { BrowserRouter, Routes, Route, Link }
  from 'react-router-dom';
function App() {
  return (
    <BrowserRouter>
      <nav>
        <Link to="/">Home</Link> |
        <Link to="/about">About</Link>
      </nav>
      <Routes>
        <Route path="/" element={<Home />} />
        <Route path="/about" element={<About />} />
      </Routes>
    </BrowserRouter>
  );
}
```

7 Redux

- Redux for global state management.
- Use createSlice and configureStore.
- Access state with useSelector.

```jsx
// store.js
import { configureStore, createSlice } from '@reduxjs/toolkit';

const counterSlice = createSlice({
  name: 'counter',
  initialState: { value: 0 },
  reducers: {
    increment: (state) => { state.value += 1 },
    decrement: (state) => { state.value -= 1 }
  }
});

export const { increment, decrement } = counterSlice.actions;

export default configureStore({
  reducer: { counter: counterSlice.reducer }
});
```

8 Deployment

- Build your app for production.
- Deploy to any static hosting.
- Popular: Vercel, Netlify, GitHub Pages.

```bash
# Build for production
npm run build

# Preview build locally
npm run preview

# Deploy to Vercel
vercel

# Deploy to Netlify
netlify deploy --prod

# Deploy to GitHub Pages
npm run deploy
```

Pro Tips

- ✅ Keep components small and focused
- ✅ Use meaningful names
- ✅ Use ESLint & Prettier
- ✅ Optimize performance
- ✅ Write tests for important parts

Useful Commands

```bash
npx create-react-app my-app   # Create a new project
npm start                     # Start development server
npm run build                 # Build for production
npm test                      # Run tests
npm run eject                 # Eject configuration (one-way)
```

Common Imports

```jsx
import React from 'react';
import { useState, useEffect } from 'react';
import { BrowserRouter } from 'react-router-dom';
import { configureStore } from '@reduxjs/toolkit';
import axios from 'axios';
```

Folder Structure (Example)

```
src/
├── components/
├── pages/
├── hooks/
├── redux/
├── services/
├── App.js
└── index.js
```